# Exercises

**March 2003**

# Online Software

## Training

**Version 1.9**

ATLAS DAQ Technical Note 149
http://atddoc.cern.ch/Atlas/Notes/149/Note149-1.html

This document has been prepared with Release 5.5 of the Adobe FrameMaker® Technical Publishing System using the User's Guide template prepared by Mario Ruggier of the Information and Programming Techniques Group at CERN. Only widely available fonts have been used, with the principal ones being:

| | |
|---|---|
| Running text: | Palatino 10.5 pt on 13.5 pt line spacing |
| Chapter numbers and titles: | AvantGarde DemiBold 36 and 24 pt |
| Section headings | AvantGarde DemiBold 20 pt |
| Subsection and subsubsection headings: | Helvetica Bold 12 and 10 pt |
| Captions: | Helvetica 9 pt |
| Listings: | Courier Bold 9 pt |

# Outline

This document has been prepared with Release 5.5 of the Adobe FrameMaker® Technical Publishing System using the User's Guide template prepared by Mario Ruggier of the Information and Programming Techniques Group at CERN. Only widely available fonts have been used, with the principal ones being:

| | |
|---|---|
| Running text: | Palatino 10.5 pt on 13.5 pt line spacing |
| Chapter numbers and titles: | AvantGarde DemiBold 36 and 24 pt |
| Section headings | AvantGarde DemiBold 20 pt |
| Subsection and subsubsection headings: | Helvetica Bold 12 and 10 pt |
| Captions: | Helvetica 9 pt |
| Listings: | Courier Bold 9 pt |

# Chapter 1

# Introduction

**What is this document about?**

This document presents a series of practical programming exercises intended for people wanting to develop detector or system specific software using the ATLAS Online Software as a framework. It does not provide user training on how to run the ATLAS Trigger-DAQ system.

**What is the Online Software?**

The Online Software is responsible for the overall experiment control, including run control, configuration of the Trigger-DAQ system and management of data taking partitions. The Online Software also includes the online monitoring infrastructure and graphical user interfaces used for control and configuration, and the means for handling distributed information management including database management and tools. It does not contain any elements that are detector specific as it is to be used by all possible configurations of the DAQ and detector instrumentation.

This chapter gives a general overview of the ATLAS Online (formerly known as DAQ Prototype -1 Back-End) software training exercises.

In these exercises you will be shown how to develop detector or system specific software using the Online Software as a framework. Four small exercises, described below, are to be made. In the first exercise, you will develop a read-out crate (ROC) controller. For the second exercise, you will develop a graphical panel capable of visualising information provided by the crate controller. In the third one you will develop a test for the configuration previously built. Afterwards you will develop an event sampler example and a monitoring task example using the On-Line Monitoring package. In order to follow these exercises you need to have a basic knowledge of the following subjects:

- unix environment (Bash shell, X Window System)
- basic object oriented concepts (object, method attribute)
- C++ (basic syntax and constructs)
- programming tools (editor and make)

This tutorial assumes you are going to perform the exercises on a linux or solaris platform using the bash shell.

**Crate controller** This part of the exercises explains how to develop a simplified ROC crate controller in C++ based on the Run Control controller skeleton. You will learn how a controller operates according to the standardized finite-state-machine.

**GUI panel** This part of the exercises shows you how to develop a simple panel in Java to visualize and track the value of a parameter of a module in the ROC crate. The panel will be integrated with the standard Online Integrated GUI.

**On-Line Monitoring** This part of the exercises shows you how to develop an event sampler in C++ and a monitoring task in java. This is useful to anyone going to implement an event sampler application that is responsible for supplying events to the event distribution sub-system or to develop a monitoring task that reads events from the event distribution.

**Online Histogramming** This part of the exercise explains how to use the Online Histogramming subsystem. You will learn how to write a histogram provider and a histogram display using C++.

**Test development and diagnostics** This part of the exercises demonstrates how to write a test for the VME module in a crate and integrate it so it can be used by the online diagnostics component.

**Resource Manager** This part of the exercises is dedicated to the usage of the Resource Manager, explaining how to ask for resources, use resources and free them, using the Resources Manager library.

**Installation of Online SW release** This document refers to the training prepared for online-00-19-00 release of the Online Software. To use training, you have to have Online Software release installed and configured. Release online-00-19-00 is available for the following platform / compiler combinations:

- Linux RedHat 7.3 / gcc-2.95
- Sun Solaris 2.8 / CC-5.2
- LynxOS 3.0.1 / g++-2.9-98r2

For the use of the Online Software and its training, it is recommended that you use **bash** shell. However it is also possible to use the **[t]csh**. In the following document it is assumed that the **bash** shell is used. Start the **bash** shell.

```
> bash
```

Software is available for download from ***http://atlas-onlsw.web.cern.ch/Atlas-onlsw/download/download_page.htm***. When using a local installation of the release, source the generated setup.sh script in the directory where the Online Software has beed installed to set up the environment:

```
> source ./setup.sh
```

If you have access to afs, you can use public installation of the release from /afs/cern.ch/atlas/project/tdaq/cmt. To set up the release, you have to source the official setup script having as agrument the release name. For example, to setup release online-00-19-00 use:

```
> source /afs/cern.ch/atlas/project/tdaq/cmt/bin/cmtsetup.[c]sh
online-00-19-00
```

When using the Online SW, in order to be a bit more independent of others using the same distribution (your local install, or the public AFS version), it is possible to make your own IPC Reference File (see the Online Software FAQ on the Online Software web site for an explanation of what an IPC Reference file is). To do this, setup the evironment as described above and run the following command:

```
> export
IPC_REF_FILE="/some/new/path/that/you/choose/ipc_root.ref"

> ipc_server -i $IPC_REF_FILE &
```

You can then always use your ipc_root_ref file just by setting the environment variable **IPC_REF_FILE** to its location.

**Installation of Training package**

The training package can be copied from the installed Online software release, **`${TDAQ_INST_PATH}/share/data/training`** directory.

To install the training exercises using an installed Online software release just copy the **`training`** directory (with all the subdirectories and files) into a directory of your choice. For example, to copy the training exercises from the release of the Online software available via AFS use the following command:

```
> cp -r ${TDAQ_INST_PATH}/share/data/training .
```

**Training Documentation**

The training documentation can be found in the **`training_doc.pdf`** and **`training_doc.ps`** files in the **`${TDAQ_INST_PATH}/share/doc/training`** directory of the installed Online software release.

**Source Code**

The directory tree containing source code of the skeletons and templates used for the exercises is structured as follows:

```
training/
            controller/  # crate controller exercise
            panel/       # GUI panel exercise
            diagnostics/ # diagnostic test exercise
            databases/   # holds partition database file
            monitoring/  # monitoring test exercise
            histogramming/   # histogramming test exercise
            resources/   # resource manager test exercise
```

The **`training`** directory is the root directory of the training exercises and the path of this directory is referred to by the environment variable **`MY_PATH`**. When you have copied the training exercises, source the configuration script:

```
> cd training

> source ./.training.sh
```

This script sets all the environment needed to build the training code against the release on your platform and also sets the environment variable **`MY_PATH`** which is

referred to in this tutorial. Note for it to be set properly you must be in the training directory when you source the setup script.

**Solutions**

The completed working solutions for each of the exercises are available in the **`solution`** directory below each exercise directory:

```
${MY_PATH}/
            controller/solution
            panel/solution
            diagnostics/solution
            monitoring/cpp/solution
            monitoring/databases/solution
            monitoring/java/solution
            histogramming/raw_provider/solution
            histogramming/root_display/solution
            resources/solution/cpp
            resources/solution/databases
```

**Example configuration**

To perform the exercises, an example configuration has been defined (Illustration 1.0). This configuration is a simple partition that can be simulated on the computer being used for the tutorial. The partition represents a detector made of a single Read-OutCrate crate. The crate contains a single module. The module has one important parameter associated with it - a counter that will be the primary interest of the exercise.

A partition of this format has been defined in a database and is available for use in the exercises. The database files that contain the definition of the partition are held in the **`databases`** directory:

**`${MY_PATH}/databases/`*`partition_name`*`.data.xml`**

**`${MY_PATH}/databases/`*`partition_name`*`.hw.data.xml`**

**`${MY_PATH}/databases/`*`partition_name`*`.sw.data.xml`**

Where ***`partition_name`*** is **`train_01`**.

The information is split accross the three files according to the following schema (see the "Configuration Database User's Guide" for more information):

- software database: software objects, resources, programs, environment and parameters;
- hardware database: workstations, cpu boards, detectors, crates, modules and parameters;
- main database: configuration including used schema and data files, partition, applications (including run control and sampling applications),

event sampling criteria, environment and parameters.

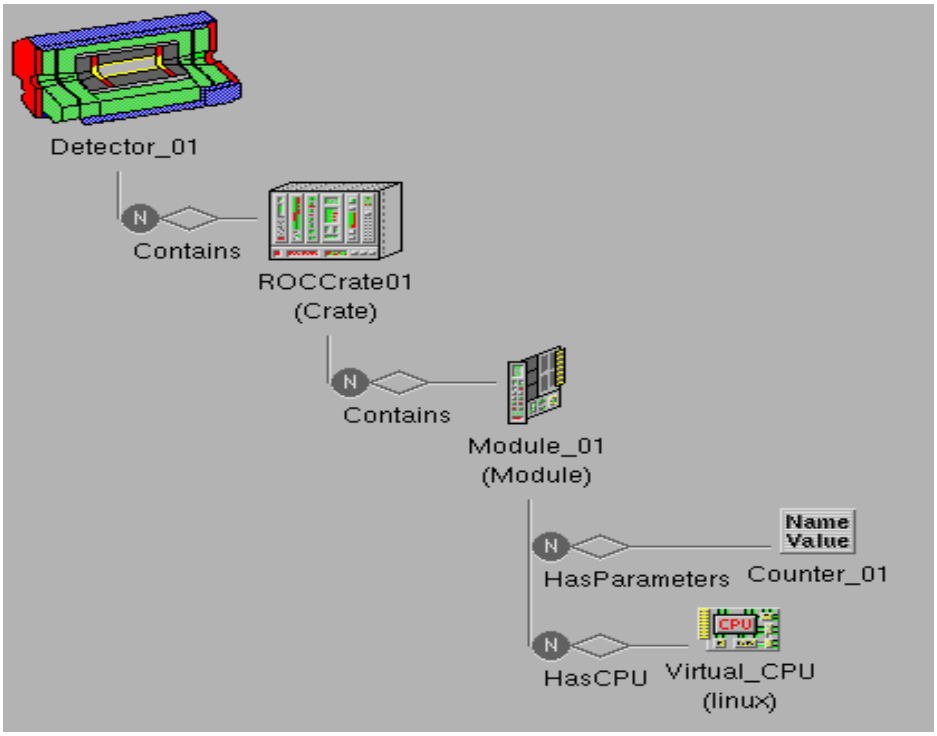



Illustration 1.0 Example partition

**Setting up the example database**

The example databases need to be modified according to your local installation. It is necessary to ensure that the host on which you will run the training exercises is defined inside the database. Workstations (or PCs) are defined in the hardware repository datafile **`train_01.hw.data.xml`** which is also in the databases subdirectory. Inside this database file, a Workstation object with the identity *MyWorkstation* is defined. You must modify the *Name* attribute of *MyWorkstation* to set it to the host name of your own workstation. Inside the database file, a CPU_Board object with the identity *Virtual_CPU* is defined. You must also change the *Name* attribute of *Virtual_CPU* to set it to the host name of your own workstation. Check that the *BinaryTag* and *RemoteLoginCommand* attributes of MyWorkstation and Virtual_CPU are also correct for your local machine (default values for these attributes as 'i686-rh73-gcc295' and 'ssh' must be OK for the most of Linux boxes).

To edit the configuration database you can either use your favorite editor and modify the XML directly (be sure to make a copy of the file first!) or use the graphical database editor **`confdb_gui`** utility. This utility takes the full path of the partition database file (or its relative path to where you are) after the "-p" command line flag. To get the full path of the main partition file use the **`confdb_get_data_filename.sh`** script as in:

```
> confdb_gui -p 'confdb_get_data_filename.sh'
```

If you are in the **`databases`** directory then the command is simply

```
> confdb_gui -p train_01.data.xml
```

When the database files are loaded into the **`confdb_gui`**, there may be warnings appearing in the status window. This happens because the database editor, after

loading each database file, reports any objects which are referenced but nor yet found. However there should be no more warnings appearing after the last "Reading X objects from data file YYYY ..." message.

To edit the workstation object *MyWorkstation* with the **`confdb_gui`**, use the item "Hardware" on the "Edit" menu. Left click with the mouse on the *MyWorkstation* object (note if the left click does not do anything, then try the right click and select modify on the popup menu). The attributes of the object appear. To edit the "Name" attribute, left click on the current value until a cursor appears and type in the name of your workstation. To change the "BinaryTag" attribute, left click on the current value and choose the appropriate option from the list.

To edit the CPU_Board object *Virtual_CPU*, open out the tree of objects as shown in Illustration 1.0. Do this by clicking the right mouse button over the detector object and selecting the "Show relationships" item of the popup menu. Do the same thing to the oject that has just appeared (Crate) as so on until you see the icon for the Virtual CPU object. Edit this object as you did for the workstation object.

When the changes are done then close that window, and go back to the window entitled "Configuration Databases Editor", and showing the loaded files. There select the item "Save" on the "File" menu. Exit from the database editor.

**Verify database contents**

After performing all the changes you must verify the database contents. You can verify the configuration database contents with **`confdb_check_data`** utility, using the following command line:

```
> confdb_check_data
```

Be sure that the utility does not report any errors before continuing (warnings can be ignored.)

**More examples and documentation**

You can see more example applications that use the Online software here:

```
> $TDAQ_INST_PATH/share/example
```

To see further details of the APIs used in these exercises you can look at the user manuals for each component that are available from the component web pages of the Online Software website at:

***http://atlas-onlsw.web.cern.ch/Atlas-onlsw/***

**Some good advice**

Don’t try to skip the first exercise, the other exercises depend on its successful completion.

# Chapter 2

# Crate Controller

This part of the exercise explains how to develop a simplified ROC crate controller in C++ using the Run Control controller skeleton. You will learn how a controller operates according to the standard finite-state-machine and how to access other Online software components from a controller.

## The run-control system

The run control is one of the software components of the ATLAS Online software. It controls data-taking activities by coordinating the operation of the DAQ sub-systems, Online software components and other systems. It has user interfaces for the shift operators to control and supervise the data-taking session and software interfaces with the DAQ sub-systems and other Online software components. Through these interfaces the run control can exchange commands, status and information used to control the DAQ activities.

Through the user interface the run control receives commands and information describing how the user wants the experiment to take data. It allows the operator to select a system configuration, parameterize it for a run and start and stop the data taking activities.

The run control system operates in an environment consisting of multiple partitions that may take data simultaneously and independently. Each copy of the run control is capable of controlling one partition.

The run control needs to send commands to the other systems in order to control their operation and to receive change of state information. The external systems are autonomous and independent of the run control so their detailed internal states remain hidden. If a system changes state the run control reacts appropriately, for example, stopping the run if a detector is no longer able to produce data. The run control interacts with a dedicated controller for each sub-system.

**Controller**

A controller receives commands from the outside world. Commands cause a controller to execute actions which potentially change the state of the controlled apparatus. The state of the apparatus is published by the controller to make it "visible" to the outside world. A controller can also react to local events occurring in the apparatus under its responsibility (for example buffer overruns). Typically its reaction will be to execute some actions and potentially change its visible state.

A controller uses other Online components to fulfill specific functionality:

- its parameters and relationships with other controllers are retrieved from the configuration database,
- it is notified of the state of its child controllers via the Information Service (IS). It also publishes its own state information in the IS,
- it produces MRS error messages to inform other programs if errors occur.

The interaction between a controller and the other Online software is shown in Illustration 2.1.

**Controller state machine**

The behaviour of a controller is modelled by a state machine (see Illustration 2.2). The state machine represents the state of the apparatus under its control and how it reacts to commands.

Although the status of each piece of controlled apparatus is modelled by the same set of states, the actions required to cause the apparatus to change from one state to another will be different. The controller skeleton has been designed to be a general template. Developers of the various controllers in different parts of the experiment customise the behaviour of their particular controller by adding code to implement the required behaviour within this generalised framework.

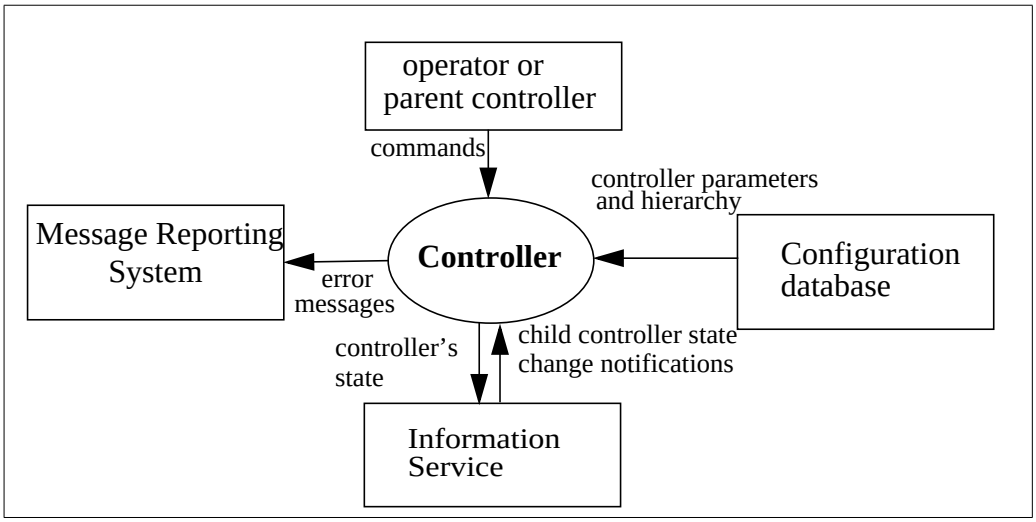



Illustration 2.1 Interactions between a controller and other Online software components

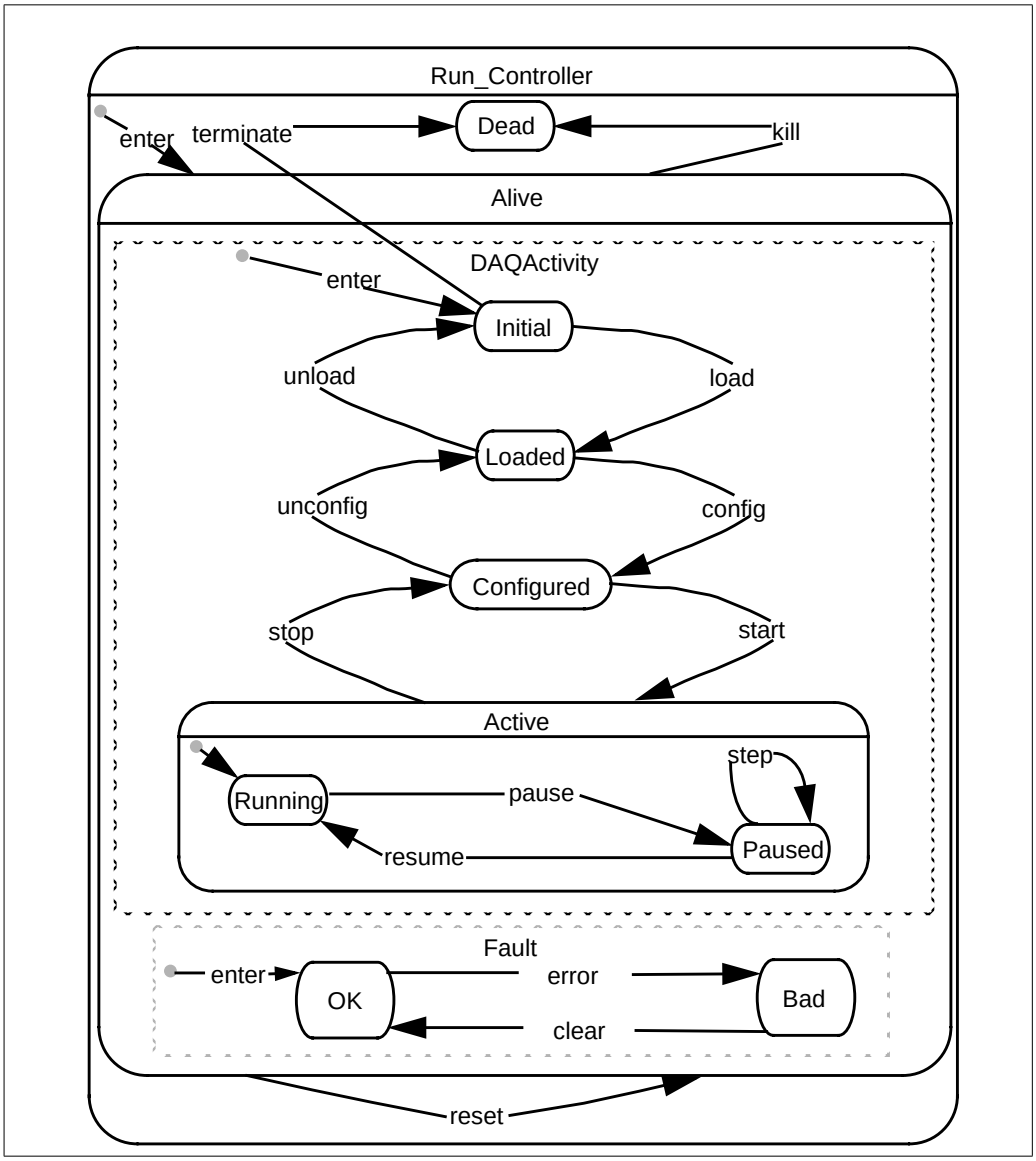



Illustration 2.2 Simplified controller state machine

### Usings Threads In The Controller

In an additional exercise you can experiment with continuous actions to be perfomed while the controller has a certain state. This can be perfomed either by the use of periodic alarms or by dispatching an independant thread.

The solution using periodic alarm has significant advantages: You can stay in the single threaded mode, which is gerally speaking a considerable safer environment. In cases where a periodic action has to be taken, this is the solution which technically better fits the problem. A typical example would be the regular updating of an IS information.

Starting seperate threads leads to more complex application designs. While a certain class of applications can profit from multithreading, in some cases synchronisation between the task can introduce a significant overhead and minimises the gain.

This consideration is also important for the libraries used by the application. They might contain no synchronsiation mechanisms either due to design or performance reasons. This is a common feature of complex libraries and also the Online Software.

Having said all that, there are cases where the use of threads is adequate. Extending the controller exercise, an example is proposed where an additional thread is detached to cycle the foreground (grid) colour of the workstation's display background, while the controller is in the running state.

The example uses the thread wrapper provided with the CORBA package. It provides a simple solution to dispatch threads. More complex design will most probably require the use of a more powerful package, as the thread package from the dataflow group.

**ROC crate controller example**

In this part of the exercise you will use the controller skeleton to develop an example ROC crate controller. Obviously, we do not have a true ROC crate available so the actions performed on the crate (apparatus) and the events it returns are simulated using software which runs on the computer used for the training.

The example controller operates within a partition made of one ROC crate which contains a single module. The module needs to be sent commands to be properly configured before a run. To simulate sending the module commands, you are asked to send commands to the X Window Server of the training computer that set the background colour of the screen. The commands should be sent when entering the various states of the controller's state machine as shown below:

– *Initial* - white

– *Loaded* - yellow

– *Configured* - magenta

– *Running* - green

– *Paused* - grey

The module has a counter which should be initialised and monitored while in the Running state. The counter value should be published in the Information Service so that it can be viewed by the operator. The value used to initialise the counter must be retrieved from the database when the controller is started. To simulate monitoring the counter during the run, a regular timer will be created and at each timer interval (3 seconds) the counter should be incremented and then published in the Information Service.

An MRS message (ROC-start / stop) should be sent on entering / exiting the Running state and when the controller is configured.

Illustration 2.3 shows all the actions associated with each state and transition of the controller's state machine.

Every transition in the diagram has an associated action. The action is a method that should return an integer value. If the value returned is zero then the transition is considered to have completed succesfully. If it returns a non-zero value then it is considered to be an error so the controller makes the transition but also enters the Bad state and sends an MRS error message reporting the problem.

In addition to transition actions, every state has an associated entry and exit action. These are executed every time the state is entered or exited, regardless of the event which caused the transition. State entry and exit actions are assumed to complete successfully and do not return a value. Each of these actions (transition and state entry and exit) can be user-defined.

A signal handler can be defined that is called when a signal is passed to the controller by the host operating system. The signal handler is used in this example exercise to execute the activeExit() method if controller exits (i.e. when the KILL signal is received).

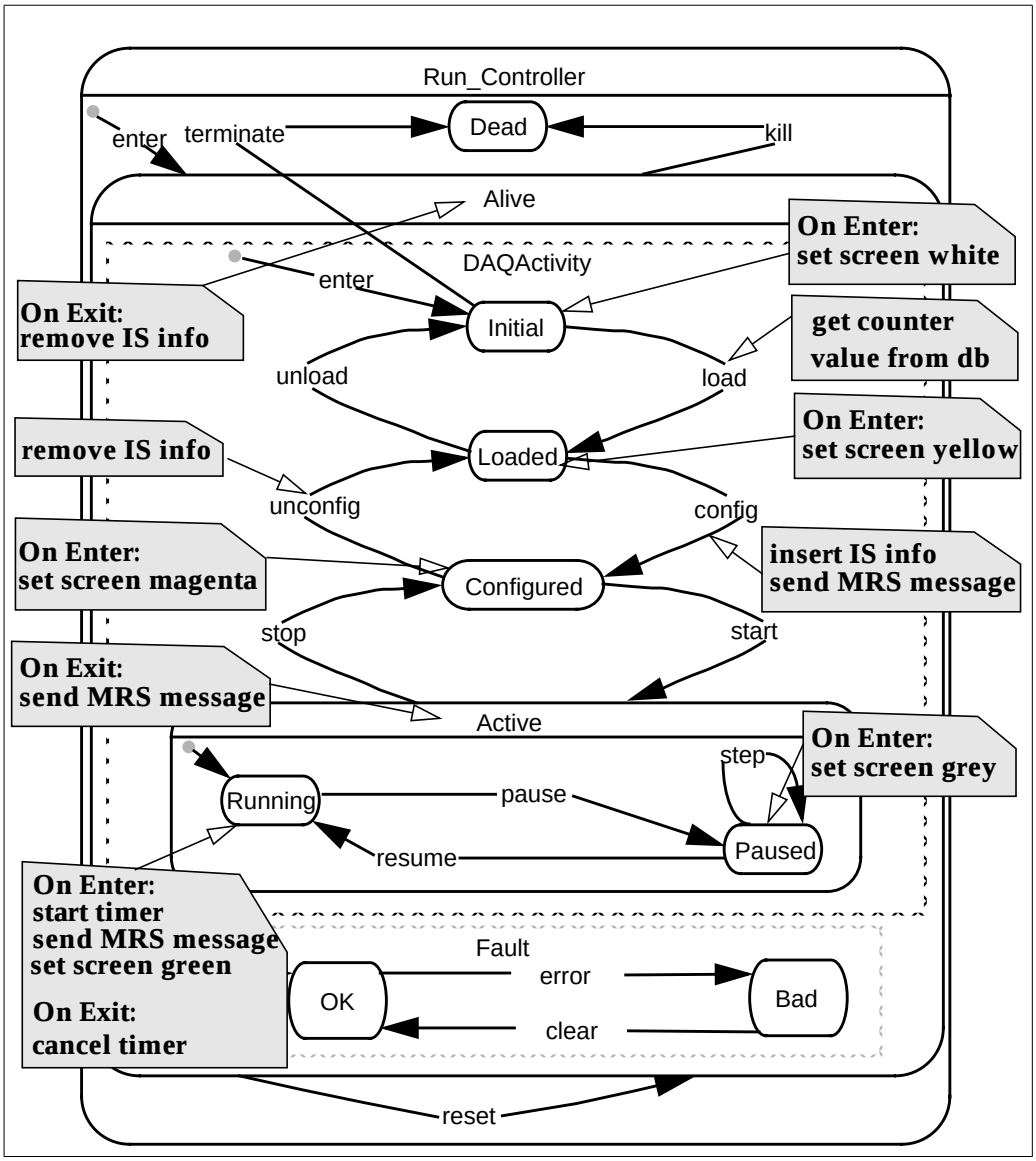



Illustration 2.3 Actions to be performed by controller

**rc_interface class**

The rc_interface class encapsulates the controller skeleton. It has virtual methods defined for all state entry, exit and transition points. For example the entry method of the Running state is called enterRunning(), the exit method is exitRunning() and the action method for the pause transition is pauseaction(). By default, all methods are empty and action methods return zero (i.e. complete successfully). No parameters are required for any of the methods.

To build a controller you must define a class that inherits from the rc_interface class and overload the appropriate methods to perform the necessary actions.

For this exercise, such a controller class has already been defined, called RCCrateExampleUser (Illustration 2.4), but a few commands are missing from its methods which you will need to complete. By studying the diagram on the previous page and the code that already exists in other methods of the class, it should not be to difficult to add the missing commands.

**Accessing the source code**

The source code for the controller is held in the **`controller`** subdirectory. These are the important files:

**`rc_crate_example_user.h`** RCCrateExampleUser definition

**`rc_crate_example_user.cxx`** RCCrateExampleUser declaration

**`rc_crate_example_ctrl.cxx`** controller main program

**`Makefile`** makefile to compile and link the controller

To browse and modify the source code, open the **`rc_crate_example_user.h`** & **`rc_crate_example_user.cxx`** source files in your favourite editor (e.g. nedit).

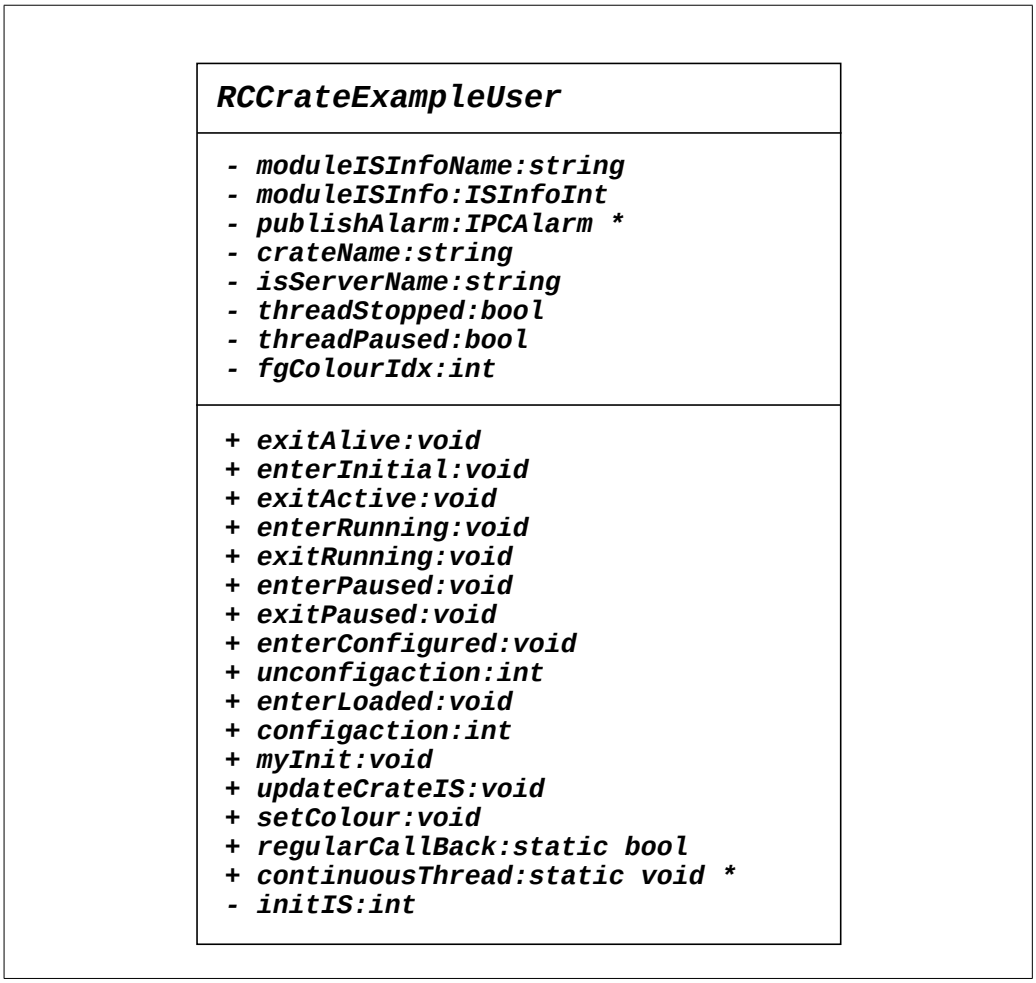



Illustration 2.4 RCCrateExampleUser class: attributes and methods

### Setting the screen colour

To simulate sending commands to a module or device, the controller should set the colour of the background of the computer screen. A method called, setColour, has been added to the RCCrateExampleUser class. This method is called from most state entry methods (see Illustration 2.5). As defined in the specification for the controller (see See “ROC crate controller example” on page 18.), the screen background should be set to yellow when entering Loaded.

- **Modify the name for the display to your own in the `XSETROOT_CMD` definition at the beginning of the `rc_crate_example_user.cxx` file. Verify the path used in the command for the `xsetroot` binary is correct on your machine (Do the same for `XSETROOT_CMD_FG`, it shall be used later for the thread example).**
- **Modify the source code (.cxx & .h) to overload the Loaded state entry method and set the screen background to yellow.**

### Sending MRS messages

Controllers send MRS messages to inform other sub-systems and the human operator of any important occurrences inside their apparatus. MRS messages are not the equivalent of a print statement and should not be used as a debugging tool (for this it is better to use the cout stream.)

The controller skeleton uses MRS internally for such purposes as sending a message when the controller enters the Bad state. It opens an MRS stream during initialisation and this stream is made available to controller developers via the rcMRSstream attribute of the rc_interface super-class.

The specification for the controller says an MRS message (ROC-start / stop) should be sent on entering / exiting the Running state and when the controller is configured. The code provided for the exercises does this already except for sending the ROC-start message when entering the Running state (see Illustration 2.6). The ROC_start message should have the following attributes:

Message name *crateName*_START

Message severity Information

Message text *crateName*-crate started operation

Message qualifier ROC

- **Modify the source code to send the ROC-start MRS message when entering the Running state.**

*rc_crate_example_user.h:*

```
virtual void enterConfigured();
```

*rc_crate_example_user.cxx:*

```
void RCCrateExampleUser::enterConfigured() {
setColour("magenta"); }
```

Illustration 2.5 Example entry method

*rc_crate_example_user.cxx:*

```
mout() << crateName+"_STOP" << MRS_INFORMATION
<<MRS_TEXT(string(crateName).append("-crate stopped
operation")) << MRS_QUALIF("ROC") << ENDM;
```

Illustration 2.6 Example of sending an MRS message

**Publishing IS information**

Controllers publish IS information to inform the other sub-systems and the human operator of apparatus specific information that may change during a run. For example, a ROC module may publish counter values representing the number of event fragments treated since the start of run, how many have been rejected by the trigger etc. Such information should be updated at regular intervals but not at a high frequency because the IS is not a real-time facility. An update interval of several seconds is normally suitable.

The controller skeleton uses the IS internally to publish it's state information. It creates an ISInfoDictionary object during initialisation and this dictionary object is made available to controller developers via the is_dict attribute of the rc_manager class from which rc_interface itself inherits.

In order to publish a piece of information in the IS, it must first be inserted into a server. This is done during the Configure transition (see Illustration 2.7).

The specification for the controller says a counter value for the module in the ROC crate should be published every 3 seconds while in the Running state. A timer facility is provided by the ILU package on which IPC is built and this is available in the skeleton via the createTimer and cancelTimer methods. These methods are used in the Running state entry and exit methods of the example controller to manage a 3 second timer. When the timer fires, the callback method updates the information in the IS server with the latest value of the counter.

Finally the information is removed from the IS server during the unconfigure transition. However, if the controller is forced to reset (i.e. receives a reset command) or exit (i.e. receives a kill command) then the unconfigure transition will not be made and the information will remain in the IS server. To avoid this situation, the code to remove the IS information has also been added to the exit method for the Alive state.

- **Modify the source code to remove the module counter information from the IS server when performing the Unconfigure transition.**

**Modify the database**

The **`train_01.hw.data.xml`** database file, from the **`databases`** subdirectory, contains the hardware repository for the configuration presented earlier. As mentioned in the introduction chapter, this database file should be modified to include the hostname of your own workstation. Please refer to the section "Setting up the example database" in the introduction chapter for how to do this.

***rc_crate_example_user.cxx:***

Initialise a timer:

```
publishAlarm =
rc_commManager::createTimer(3,&regularCallBack, this);
```

Cancel a timer:

```
rc_commManager::cancelTimer(publishAlarm);
```

Insert an IS information:

```
rc_manager::is_dict->insert(moduleISInfoName.c_str(),
myISInfo);
```

Update an IS information:

```
rc_manager::is_dict->update(moduleISInfoName.c_str(),
myISInfo);
```

Remove an IS information:

```
rc_manager::is_dict->remove(moduleISInfoName.c_str());
```

Illustration 2.7 Example timer, IS insert, update and remove actions

### Retrieving information from the Configuration Database

The specification for the ROC crate controller says that it should retrieve the value used to initialise the counter from the configuration database when the controller is started (Parameter class, Counter_01 object in partition_name.hw.data.xml database). In this example, we will use the Data Access Library (DAL) to access the database.

A method called initIS (see Illustration 2.8) has been added to the controller's class which retrieves the counter value from the database and uses it to initialise the information to be published in IS. Once the database has been initialised, it is necessary to have some basic knowledge of the schema in order to find the required information. The schema determines how the application should navigate through the relationships of the database class in order to retrieve the counter value:

- get physical partition
- find the crate using its name as a key
- find the first module contained in the crate
- find the first parameter of the module
- get the name and the value of the parameter

The value of a parameter can be retrieved from the databases using the *get_value*() method of the *ConfdbParameter* class which returns a character string. The character string can be converted to an integer value using the standard *atoi* C library routine.

- **Modify the initIS method of the RCCrateExampleUser class to retrieve the value of the counter parameter from the database, convert it to an integer and store it in the moduleISInfo attribute.**

### How to build the controller

You should now have all the source code necessary for the example ROC crate controller. Change to `controller` subdirectory. You can now build the controller using the makefile provided:

```
> make          # compile and link the controller
```

*rc_crate_example_user.cxx:*

```
ConfdbConfiguration confDb((const char *) 0, (const char *) 0,
     (ConfdbConfiguration::CreateObjectFN) 0,
     (ConfdbConfiguration::InitFN) 0);
if (confDb.get_status() != ConfdbDataFlowConfiguration::Success){
     cerr << " ERROR: Failed to load database" << endl;
     return -1;
} else {
     cout << "RCCrateExample: initialized database" << endl;
}

//find the crate in the database
ConfdbCrate * myCrate = confDb.find_crate
     (RCCrateExampleUser::crateName);
if (myCrate == 0) {
     cerr << "RCCrateExample: did not find the crate " << crateName
          << endl;
     return -1;
     }
cout << "RCCrateExample: found crate" << myCrate->get_name() << endl;

//try to get the first module in the module list of the crate
const list<ConfdbModule *> moduleList= myCrate->modules();
ConfdbModule * module = *(moduleList.begin());
if (module == 0) {
     cerr << "RCCrateExample: No modules defined in crate " <<
          crateName << endl;
     return -1;
}
string moduleName(module->get_name());
cout << "RCCrateExample: found module" << moduleName << endl;

//try to get the first parameter in the parameter list of the module
const list<ConfdbParameter *> paramList= module->parameters();
ConfdbParameter * counter = *(paramList.begin());
if (counter == 0) {
     cerr << "RCCrateExample: No parameters defined in module " <<
          moduleName << endl;
     return -1;
     }
cout << "RCCrateExample: found counter" << endl;
string counterName(counter->get_name());

//set start value for the counter from the parameter
//initialize the IS info
// add code to get parameter from DB, convert to integer and store in
moduleISInfo here
```

Illustration 2.8 code extract from the initIS method

**How to test the controller**

When your controller compiles and links correctly, you can test it as part of the small partition introduced at the start of the tutorial. The play_daq script can be used to start the partition but it needs to have a few environment variables set first so make sure your have the following variables defined in your environment:

TDAQ_DB_DATA This defines the data file holding the partition. It should point to the partition data file in the databases directory (See "Source Code" on page 9.), for example:

```
> export TDAQ_DB_DATA=${MY_PATH}/databases/
partition_name.data.xml
```

Where *partition_name* is the name of the partition, **`train_01`**.

Once you have verified your environment, you can start the partition by calling the play_daq script but note that you should start a new terminal window to do this (i.e. do not run **`play_daq`** from the console):

In the other window start **`play_daq`** script:

```
> export DISPLAY=your_display:0.0     # set your display
```

and make sure the controller can write to the screen (e.g. use xhost +).

```
> play_daq partition_name no_obk     # start the partition without book-keeper
```

The **`no_obk`** option tells **`play_daq`** not to start the online book-keeper. This involves a lot more resources and is not useful in the context of the training. When the IGUI appears you should press *boot* to cause your controller to be started by the DAQ Supervisor. You can then send it different run control commands to change its state and put it in the *Running* state.

Verify the controller behaves according to the specification and then stop the partition by pressing *shutdown* and *exit*.

Note that in the IGUI, the MRS Panel will by default not show any messages. This is because the selection criteria for the MRS messages excludes all INFORMATION messages by default. You should set the selection so that the MRS messages from the controller can be seen and checked (selecting ALL for the subscription in the MRS panel).

**Checking IS information**

You can check if your controller is publishing the correct information to IS by using one of the programs intended for developers (not end-users) when your partition is *Booted* and in the *Running* state. The is_monitor (see Illustration 2.9) provides a basic GUI for viewing IS information:

```
> is_monitor
```

- Select the name of your partition.
- The list of IS servers will appear.
- Select DF then press **Show Info List** and the list of information items will appear
- Select your parameter and its value will be displayed.
- Wait while the partition is in the running state to see the information changing.

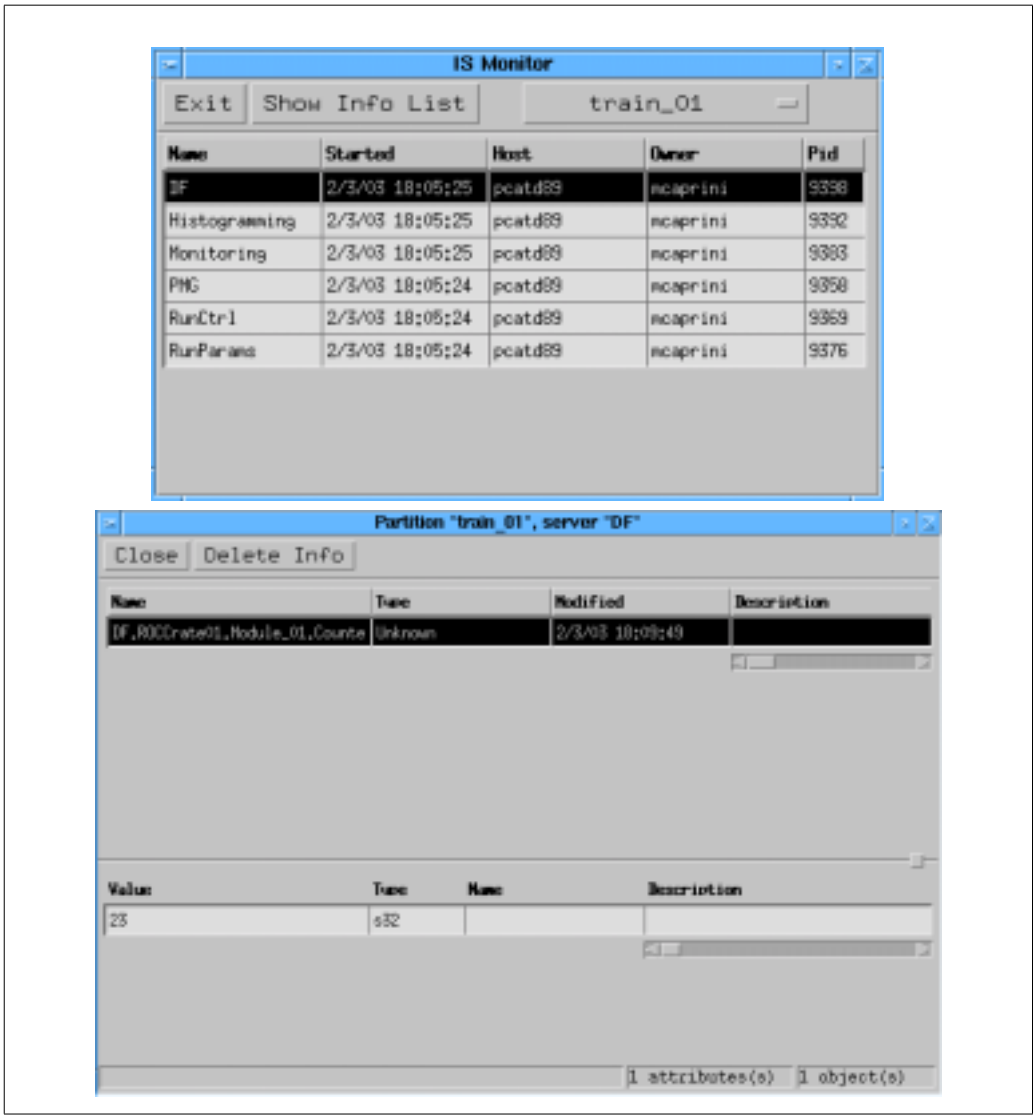



Illustration 2.9 is_monitor application

**Exercise to add an ILU thread in the crate controller**

In this extension of the controller exercise, you can add an additional thread to the controller, to cycle the foreground (grid) color of the workstation's display background, while the controller is in the running state. The thread should be started when the Running State is entered and stopped when the Running state is finished. In addition it should pause execution during Paused State.

**Thread declaration and initialisation**

- In order to make the ILU multi-threaded mode available, you have to check that a header-file include statement has been added in the header file. A method to change the foreground color and a method to describe what the thread is going to do must be implemented. They have to be declared in the public section of the RCCrateExampleUser class. The boolean variables threadStopped and threadPaused together with the integer fgColourIdx must be declared in the private section of the RCCrateExampleUser class (See Illustration 2.10).
- You have to initialize the ILU multi-threaded mode in the controller's main program (See Illustration 2.11).

In file *rc_crate_example_user.h*

```
#include <ilu/threads.h>
...
  // overloaded setColour method:
  void setColour(const string &colourFg, const string
&colourBg);
  // static routine to be dispatched:
  static void * continuousThread(void * parameter);
...

private:

...
  // communication with the thread:
  bool threadStopped;
  bool threadPaused;
  int fgColourIdx;
...
```

Illustration 2.10 Thread additions to rc_crate_example_user.h

In file *rc_crate_example_ctrl.cxx*

```
...
   unsigned status = cmd.parse(argv_iter);
   if (status) {
    cmd.error() << av[0] << ":parsing errors occurred!" << endl;
   return status;
   }

#ifdef THREAD_EXAMPLE
   IPCCore::init( true ); // Initialize in multi-threaded mode
#endif
...
```

Illustration 2.11 Initialisation of multi-threaded mode in controller

**Thread implementation**

- In the user file, the above declared methods have to be implemented. Also, when entering the Running state, a thread has to be initialized and started (See Illustration 2.12).
- Add in the *rc_crate_example_user.cxx* file communication code with the thread. An example is provided in the **`enterPaused`** method where the statement "**`threadPaused = true;`**" has already been added.
- Make sure that when the controller is initialized, **`threadStopped`** is set to **`true`** (method **`RCCrateExampleUser::myInit`** in file *rc_crate_example_user.cxx*).

**How to build the controler with the thread extension**

- In the Makefile you will have to uncomment the line that defines the THREAD variable (**`# THREAD = -DTHREAD_EXAMPLE`**).
- Run "**`make clean`**"
- Run "**`make`**"

***NOTE:*** To test the controller with the thread extension, use the **`play_daq`** program as described above (see page 28).

**Modify the parameter value in the database**

You can modify the initial parameter value retrieved from the database by using the database configuration editor:

```
> confdb_gui -p $TDAQ_DB_DATA
```

- From the **Edit** menu select **Hardware**
- In the **Hardware** window hold down the right mouse button over the detector to reveal the pop-up menu and select show relationships
- Show the relationships for the crate and module
- Click on the **Parameter** with the left mouse button to see the Parameter's attributes
- Click the left mouse button on the **Value** field and set it to your chosen value
- Select **Save** from the **File** menu of the main window then **Exit**

For your modification to take effect the controller must re-read the database contents. According to its specification, it does this during the load transition so you must use the IGUI to put it back in the Initial state.

In file *rc_crate_example_user.cxx*

```
...
void RCCrateExampleUser::setColour(const string &colourFg,const string
&colourBg) {
string command = XSETROOT_CMD_FG + colourFg + " -bg " + colourBg;
int i = system(command.c_str());
   if (i) cerr << "RCCrateExample:system call to set fg colour failed:" << i
   << " '" << command << "'" << endl;
   else cout << "RCCrateExample: fg colour set to " << colourFg << endl;
}

void * RCCrateExampleUser::continuousThread(void * parameter) {

   RCCrateExampleUser * that = static_cast<RCCrateExampleUser*>(parameter);
   cout << "---------------> Starting thread ..." << endl;
   while ( ! that->threadStopped ) {
   while ( that->threadPaused & ! that->threadStopped ) {
   usleep(100000);
   }
   if ( that->threadStopped ) {
   cout << "===============> Ending thread ..." << endl;
   return 0;
   }
   switch(that->fgColourIdx) {
   case 0:
   that->setColourFg("blue", "green");
   break;
   case 1:
   that->setColourFg("red", "green");
   break;
   case 2:
   that->setColourFg("magenta", "green");
   break;
   default:
   that->setColourFg("white", "green");
   break;
   }
   usleep(500000); // sleep 500 ms
   that->fgColourIdx = (that->fgColourIdx+1)%4;
   }
   return 0;
}
...
void RCCrateExampleUser::enterRunning() {
   //start regular IS info update when entering the Running state
   publishAlarm = rc_commManager::createTimer(3, & regularCallBack, this);
   * rcMRSstream << string(crateName).append("_START").c_str()
   << MRS_INFORMATION << MRS_TEXT(string(crateName).append("-crate started
   operation").c_str()) << MRS_QUALIF("ROC") << ENDM;
   setColour("green");
#ifdef THREAD_EXAMPLE
   // Initialize and start a thread when entering the Running state
   // if there is no running thread (resuming from Paused state).
   if ( threadStopped ) {
   threadStopped=false;
   threadPaused=false;
   fgColourIdx=0;
   ILU_ERRS((no_memory, no_resources,internal)) err;
   ilu_OSForkNewThread( (ilu_TransportInputHandler)
   RCCrateExampleUser::continuousThread, this, &err );
   }
#endif
}
...
```

Illustration 2.12 New methods for multi-threaded controller

**Detecting faults**

If you have a problem starting the partition you can use the diagnostics package (See “Diagnostics Test” on page 43.) to determine the error. The **play_daq** and Online software also produces log files for each program that is run and you use the log files to see the exact details of what was executed. The log files are written by default to this directory:

```
${HOME}/logs/<partition name>/<user>/      # log file directory
```

When you need to change the path where all the logs files are written you have to redefine the environment variable TDAQ_LOGS_PATH (general log path) and the environment variable PMG_PROCESS_LOGS_PATH (log path for the processes started by the PMG Agent) before calling play_daq as follows:

```
> EXPORT TDAQ_LOGS_PATH=<your own logs path>

> EXPORT PMG_PROCESS_LOGS_PATH=<your own logs path>
```

# Chapter 3

# GUI panel

This part of the exercise shows you how to develop a simple panel in Java to visualize and track the value of a parameter of a module in the ROC crate. The panel will be integrated with the standard Online Integrated GUI.

## The integrated GUI

The Integrated Graphical User Interface (IGUI) is one of the software components of the Online Software sub-system of the ATLAS Trigger/DAQ project. The IGUI (see Illustration 3.1) is intended to give a view of the status of the data acquisition system and its sub-systems (Dataflow, Event Filter and Online) and to allow the user to control its operation.

The IGUI interacts with many components in a distributed environment and uses CORBA interfaces for communication with other components. It has a modular design for easy integration with different sub-systems. It is implemented in Java and uses the Java Foundation Classes (JFC) for portability and swing for graphical widgets.

IGUI is a Java application (JFrame). On the left side of the frame are displayed the Main Commands and below are some major Run Parameters, such as run and event number. On the right side there are different Panels which can be chosen by clicking the corresponding tab buttons:

- **Run Parameter** panel, which is the default view, showing all the run parameters and allowing the user to set them;
- **Run Control** panel, showing the tree and status for each controller with the possibility to send commands to a particular controller;
- **DAQ Supervisor** panel, containing the DAQ Supervisor expert commands;
- **Process Manager** panel, showing the list of PMG agents and processes;
- **MRS** panel, showing all the messages received grouped in a table and allowing the user to change the filter, subscription and log control;
- **Data Flow** panel, showing the data flow configuration and data flow parameters.

Others panels could be added, displaying the status of other DAQ components or sub-systems. The aim of the exercise is to show how such a panel can be developed.

On the bottom of the main frame there is a MRS message display panel, showing all the received MRS messages.

Illustration 3.2 shows the interaction between the IGUI and other Online components. The IGUI reads the list of partitions from the Inter Process Communication (IPC) server and lets the user select one of them. In interaction with the Resource Manager server the type of the access control is decided (only status display, normal user control or DAQ expert control). The run control configuration and the data-flow configuration are read from the configuration database. The information about the sub-systems or components status (run control status, lists of Process Manager agents and of running processes, Data-Flow modules statistics) is read from the Information Service (IS) or is automatically obtained using the IS notification mechanism. The run parameters can be set by the user and are stored in the Information Service. Through the IGUI the user can send commands to the Run Control main components (DAQ Supervisor and Root Controller). The messages sent by the Message Reporting System are received and displayed by the IGUI. The user can send commands to the MRS (to change the filter or subscription criteria, to set the log control). The IGUI can be a client of the Process Manager, allowing to start auxiliary processes (monitoring tasks, bookkeeping tasks, etc.).

In order to give the developer of a new panel the possibility to use some of the classes designed to interact with different components, the on-line documentation (Illustration 3.3) of the IGUI (packages, classes, attributes, methods) can be found at:

**http://atddoc.cern.ch/Atlas/DaqSoft/components/is/online-doc/index.html**

**IGUI panel example**

In this part of the exercise you will develop a panel to display the information published and updated by the crate controller developed in the first part of the exercise.

In order to be added to IGUI, the only requirement for a panel is to extend the ***IguiPanel*** class in the package ***igui***. ***IguiPanel*** extends the class ***javax.swing.JPanel*** and defines some interfacing methods which are common for each Panel and which each Panel have to supply (e.g. a method returning the panel name).

The panel will contain only two labels, one for the parameter name (read from database) and another for the parameter value (updated by IS notification).

**RDB interface**

The panel needs from the database the information about the configuration (crate, module, parameter). The IGUI gets this information through the Remote Database (RDB) server using the ***igui.RdbInterface*** class which implements the CORBA client side for remote database access. The following methods are useful for the panel design:

- ***findPartition -*** checks that the working partition is defined in the database;
- ***getObjectList -*** gets a list of all the objects related by a relationship to an object in the database (for example all Crates contained in a Detector).

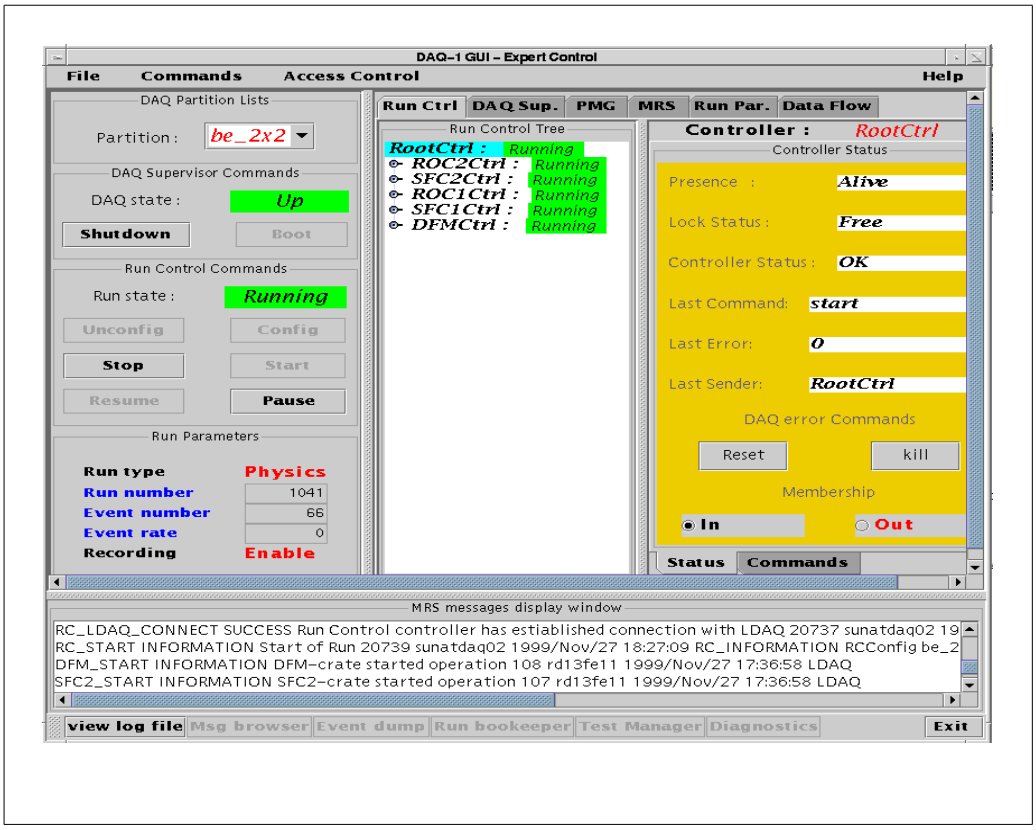



Illustration 3.1 Integrated Graphical User Interface

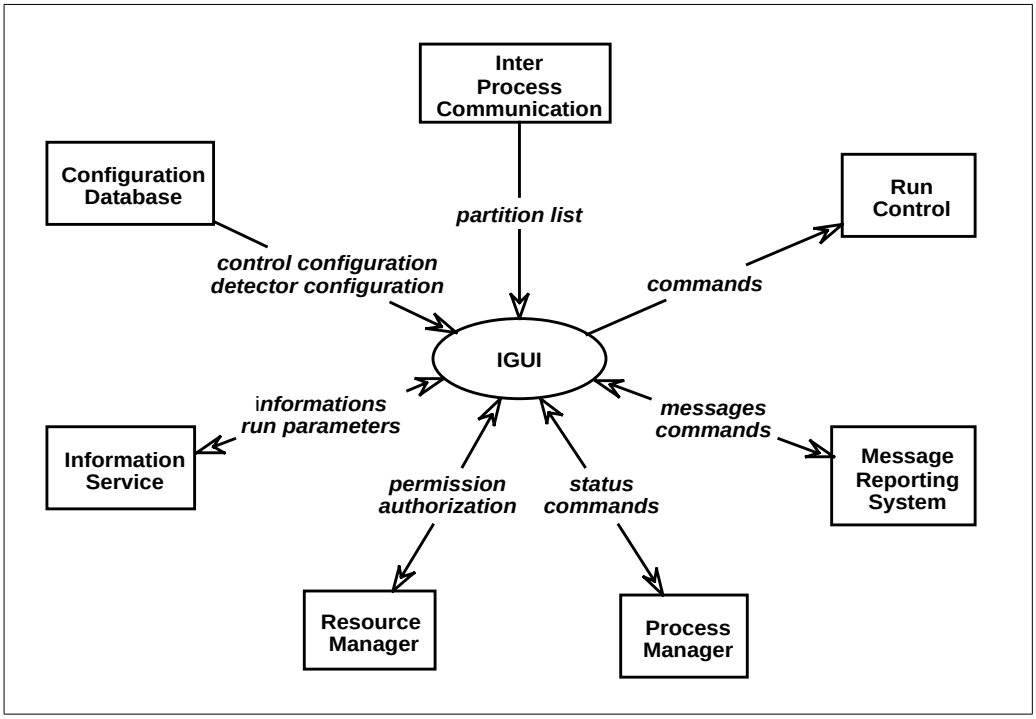



Illustration 3.2 IGUI context diagram

**IS interface**

For the interaction with the Information Service there is a special Java package, ***is***, with classes as ***AnyInfo, InfoEvent*** and ***Repository*** and the interface ***InfoListener***.

The ***Repository*** class (Illustration 3.3) contains methods to get information from the IS (***getValue***), to create new information (***insert***) and to update it (***update***) or to remove it (***remove***). In addition the subscribe mechanism is implemented (methods ***subscribe*** and ***unsuscribe***) to have the IGUI notified each time the IS information changes. The subscription method passes as parameter an object that implements the ***InfoListener*** interface. The user must define the specific actions to be done when notification occurs in the ***infoCreated, infoUpdated*** and ***infoDeleted*** methods of a class implementing the ***InfoListener*** interface. When a notification occurs, the information is passed as an ***InfoEvent*** object. The ***InfoEvent*** class has the method ***getValue*** which sets the attributes values of the information object to the values corresponding to the current event. It is possible to pass to this method either an object of the ***AnyInfo*** class or an object of the same class as the class of the object whose change is reported.

In the actual design the DAQ configuration uses six IS servers (four for the information published by Online components, Run Control, Process Manager, Monitoring and Hostogramming, one for Data-Flow sub-system and one for Run Parameters). For this exercise the Data-Flow IS server will be used (the server name is “DF”).

**IGUI panel methods**

The panel will have a constructor and two methods, one to read from database the parameter and information names and another to execute the specific action (set the text in the parameter value label) when the information is updated.

In the constructor (***RodPanel***), the following operation will be performed:

- get the partition name from the class in which panel will be inserted;
- read database to find the parameter and information names;
- add labels to the panel using a Grid Layout;
- subscribe for notification on the IS server.

The method to read from the database through RDB server (***readBD***) uses the ***RdbInterface*** class. It is supposed that in the database there is only one crate, having one module with one parameter. The following steps have to be done:

- find the Partition object;
- get the name of the detector (related to the Partition by a “UsesDetectors” relationship);
- get the name of the crate (related to the Detector by a “Contains” relationship);
- get the name of the module (related to the Crate by a “Contains” relationship);
- get the name of the parameter (related to the Module by a “HasParameters” relationship);
- information name is obtained as:

```
serverName.crateName.moduleName.parameterName
```

- use the parameter name to set the text in the parameter name label.

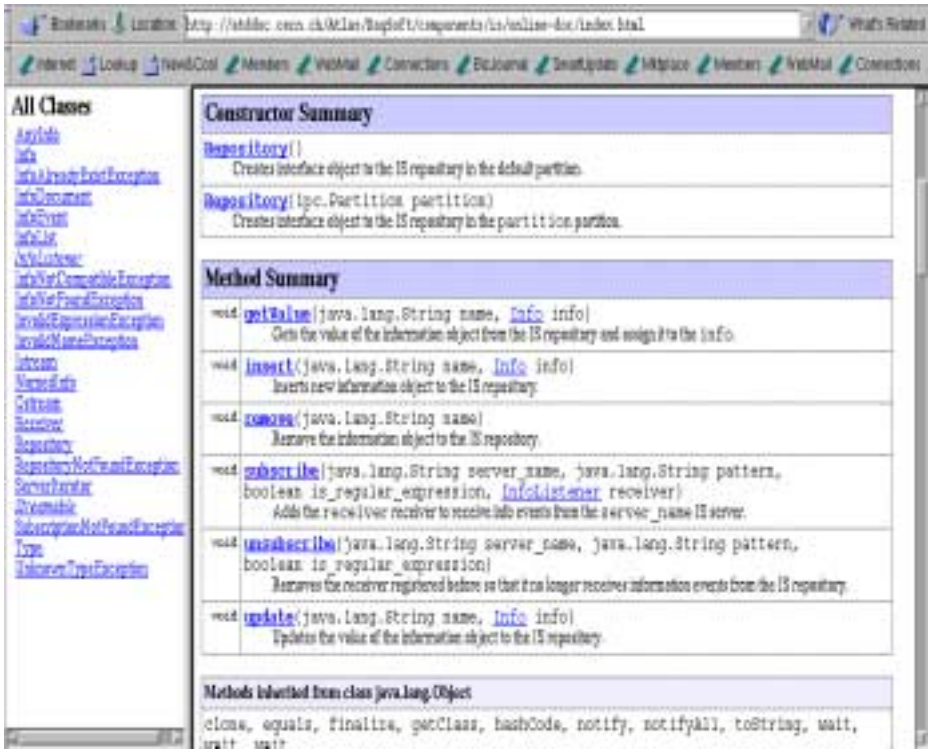



Illustration 3.3 On-line documentation for Repository class


Illustration 3.4 RodPanel attributes and methods

The method to execute the specific action when the information is updated (***infoUpdated***) will set the text on the parameter value label using the information received by the callback mechanism. The code has to:

- check if the information name is correct;
- retrieve the information from the InfoEvent;
- use the information data to set the text in the parameter value label.

**IguiPanel Interface**

Each user panel for the Igui Frame has to extend the class ***IguiPanel*** which defines some methods the user panel has to supply like a method ***panelDeselected()*** which should contain the code which should be executed, when the panel is closed.

One of the method is called ***getTabName()*** which gives back the name of the panel which should be used for the tab buttons in the IGUI frame. This method has to be supplied by you.

All the methods which are declared ***abstract*** in the class ***IguiPanel*** have to be declared, however, if there is nothing to be done, one can just declare an empty method.

**Accessing and modifying the source code**

The source code for the panel (Illustration 3.4) is held in the `panel` subdirectory. The file to edit is:

`RodPanel.java`

To browse and modify the source code, open the source file in your favourite editor.

The source code does not contain the interaction with the Information Service. If the code is compiled and tested, the panel will have the labels, but the parameter value will be not updated.

Modify the source code to add the following to the RodPanel class:

- **the subscription to IS in the constructor; (Illustration 3.5)**
- **an infoCreated method; (same as infoUpdated)**
- **an infoUpdated method; (Illustration 3.6)**
- **an infoDeleted method (empty method)**
- **an getTabName method (Illustration 3.7)**

**Compiling and testing the panel**

Verify the environment variable **PATH** includes a reference to the jdk directory (e.g. /afs/cern.ch/sw/java/XXXXX/jdk/sun-1.4.1/bin)

Verify the environment variable **`CLASSPATH`** includes ${MY_PATH}/panel, ${MY_PATH}/monitoring/java and all the jar files (ipc, is, mrs, rdb, igui, dvsgui, dvs, ed & Monitoring) held in the Online software release directory (e.g. ${MY_PATH}/monitoring/java:${MY_PATH}/panel:${TDAQ_INST_PATH}/share/lib/ipc.jar:${TDAQ_INST_PATH}/share/lib/is.jar:${TDAQ_INST_PATH}/share/lib/mrs.jar:${TDAQ_INST_PATH}/share/lib/rdb.jar:${TDAQ_INST_PATH}/share/lib/igui.jar:${TDAQ_INST_PATH}/share/lib/dvsgui.jar:${TDAQ_INST_PATH}/share/lib/dvs.jar:${TDAQ_INST_PATH}/share/lib/ed.jar:${TDAQ_INST_PATH}/share/lib/Monitoring.jar)

Change directory to the panel subdirectory and compile the code:

```
> javac RodPanel.java
```

To test the panel before including it in the IGUI we use a simple TestPanel class (see **`TestPanel.java`** file in the panel directory). This class has a ***main*** method in which the RodPanel is added to a frame. The TestPanel was automatically compiled at the same time as the RobPanel.

To test the panel in stand-alone mode (i.e. detached from the IGUI) the following steps have to be done:

- Verify the **IPC_REF_FILE** environment variable is set ;
- Start the partition (in another window):

**> play_daq partition_name no_obk**

```
is.Repository isRepository = new is.Repository (new
ipc.Partition(partition));

try {

   isRepository.subscribe(serverName, ".*", true, this);

 } catch (is.RepositoryNotFoundException ex) {

    System.out.println(" RepositoryNotFoundException in
RodPanel subscribe !");

} catch (is.InvalidExpressionException ex1) {

    System.out.println(" InvalidExpressionException in
RodPanel subscribe !");

}
```

Illustration 3.5 IS suscription

```
public void infoUpdated( InfoEvent infoEvent) {

if (infoName.equals(infoEvent.getName())) {

            is.AnyInfo ai = new is.AnyInfo();

            infoEvent.getValue(ai);

parameterValue.setText(((Integer)ai.getAttribute(0)).toString(
));

}

}
```

Illustration 3.6 infoUpdate method

```
/**
 * method to return the name for the panel in the tab button
 *   <p>
 *   @return name of panel
 */

public String getTabName() {

  String TabName = “MyPanel”;

  return TabName;

}
```

Illustration 3.7 getTabName method

- Start the TestPanel:

```
> java -Dipc.ref.file=$IPC_REF_FILE TestPanel partition_name
```

The IPC parameter is needed to establish communication.

**Testing the panel with IGUI**

To test the panel integrated in the IGUI the following steps are to be executed:)

- set the **PROPERTIES** environment variable so that the *play_daq* script will start the IGUI with your panel:

```
> export PROPERTIES="-Digui.panel=RodPanel"
```

- start the partition:

```
> play_daq partition_name no_obk
```

# Chapter 4

# Diagnostics Test

This part of the exercises explains how to develop a test application based on a C++ test template. You will learn how to make your own test repository and so integrate a new test with the online Diagnostics component.

## The Diagnostics Verification System

The Diagnostics System (DS) is one of the software components of the ATLAS Online software. It helps a human operator to initialize, test, setup and run the DAQ system without deep knowledge of its structure and functional features. The DS is designed and implemented as two separate components.

The verification component of the DS (Diagnostics Verification System) uses the tests held in the Test Manager (TM) to test the configuration and confirm functionality of any DAQ subsystem or a component. By grouping tests into logical sequences, DVS can examine any component of the system (hardware or software) at different levels of detail in order to determine the functional state of components or the entire system.

## A Test

There are two distinct phases of creating a test: the first one is to write and compile the test program and the second is to store the test in the test repository to make it available for the Diagnostics framework.

There are a couple of requirements for a proper test program. The most important one is that it has to return a valid test result. This result is passed as exit status of the program, which implies that a test program should always finish with a proper exit status. The result of the test has to comply with the POSIX 1003.3 definition and should be of type TestResult, which is defined in the TM's include file <tmgr/tmresult.h>.

```
typedef enum tmResult
{
    TmPass =          TM_PASS,
    TmUndef =         TM_UNDEF,
    TmFail =          TM_FAIL,
    TmUnResolved =    TM_UNRESOLVED,
    TmUntested =      TM_UNTESTED,
    TmUnsupported =   TM_UNSUPPORTED
} TestResult;
```

For a definition of the meaning of the results please refer to the Test Manager component documentation (ATLAS DAQ TN 66: http://atddoc.cern.ch/Atlas/Notes/066/Note066-1.html)

**Test Repository**

Test repository database stores all the information about tests. A typical test is described in a database by one instance of Test, Test4Object or Test4Class class, one instance of SW_Object class and few instances of Program class (one instance per platform). Test-derived classes describe test itself, SW_Object and Program classes describe test's implementation as for any application. All this information is used by Test Manager (via TestDAL and via Software DAL) to execute tests.

Typically, it is the developer of the test who creates all needed database objects in the Test Repository database (and probably the repository data file also). Developer can create separate repository database file with his/her own tests. This repository then shall be included in the configuration database file, so DVS and TM are able to retrieve tests for a particular objects from the configuration and execute them in the diagnostics framework.

Test, Test4Object and Test4Class classes are defined in ${TDAQ_DB_PATH}/online/schema/TestRepository.schema.xml. This schema shall be loaded with any configuration that uses Test Repository. Your partition data file train_01.data.xml already has it loaded.

For the detailed description of TestDAL and Test Repository organization please refer to the Test DAL note, published as

**`http://atddoc.cern.ch/Atlas/DaqSoft/components/diagnostics/testdal/TestDAL.html`**

**Accessing the source code**

The source code for the test is held in the diagnostics subdirectory. These are the important files:

**`test_vme_interface.cc`**
- test template;

**`Makefile`**
- makefile to compile and link the example test;

**`makefile.linux|solaris|lynxos`**
- platform specific makefiles included from Makefile;

To view and modify test's source code, open the test template file in your favourite text editor. It is shown on Illustration 4.1

**Checking IS information**

In order to simulate a test for the module we propose to check the information in the IS published by the controller. This information is the module counter. The name of the information is taken from the database and consists of the following elements:

**ISServerName.CrateName.ModuleName.ParameterName**

The controller periodically updates this information simulating the activity of the respective module. The following steps should be done in order to verify that the controller is doing this properly:

- Construct the information name

- Retrieve the information from IS
- Wait for the appropriate period of time. This period must be slightly bigger then the information update frequency. It is enough to wait for 5 seconds.
- Retrieve the same information from IS again
- Compare the values of the two information objects (they should be different)
- Return the appropriate result

Illustration 4.1 The complete test program for controller

```
 // construct information name

  string name(is_name);
  name += ".";
  name += crate_name;
  name += ".";
  name += module_name;
  name += ".";
  name += param_name;

  // get the value of information objects on is_name IS server.
  // If the information is not found then return TmFail
  ISInfo::Status        status;
  ISInfoInt             value1;

  // insert your code to retrieve the item value1 from IS here..........

  // The information for the module's parameter shall be updated each 3
seconds, so we wait 5 seconds and check it again

  sleep(5);

  // again get the value of information objects on is_name IS server.
  // If the information is not found then return TmFail
  ISInfoInt             value2;

 // insert your code to retrieve the item value2 from IS here..........

  if ( value1 == value2 )
  {
        if ( verbose ){
                cerr << "ERROR:: value was not changed during 5 seconds " <<
endl;
        }
        return TmFail;
  }

  return TmPass;
}
```

**Retrieving information from IS**

The value of an information item can be retrieved from the IS server using the findValue method:

```
ISInfo::Status        status;
ISInfoInt             value;

status = id.findValue( name.c_str(), value);
```

Compare the status returned against ISInfo::Success to verify successful completion of the operation.

- **Modify the C++ test program template to add code that retrieves the counter value from IS twice with a delay of 5 seconds between and then compares the two values.**

**Build the test**

Change directory to the diagnostics subdirectory and execute the make command to build the test binary:

```
> make
```

**Test Repository Browsing (Modification)**

The Test Repository is already created for you. There is no need to modify it. It is included in your partition database file, so you can browse and edit it if needed in database editor started by command

```
> confdb_edit_data.sh -d 'confdb_get_data_filename.sh'
```

After you load the configuration, select MyTestRepository.data.xml in the list of loaded datafiles and check all four objects defined in this repository:

two instances of Program, referring to compiled binary file for Solaris and Linux platforms: my_test_for_linux and my_test_for_solaris

one SW_Object which implements your test it refers to both Program objects.

one Test4Object - this is most interesting object. Pay attention to the following attributes:

- *object_id* - OKS ID of object you want to test. It is set to '***LDAQ@Module_01***', ID of the virtual module you intend to test with your test. This means that you are testing one particular module and this test is not applied to other instances of LDAQ class. To create a test for a class, instantiate *Test4Class* class.
- *is-a* relationship - links test object with it's implementation - SW_Object.
- *timeout* - set to default value 0. If you expect that your test may "hangs", put this to some reasonable value (in seconds)
- *host* - the name of the host on which this test is executed by TM. If empty, the default (local) host is used.
- *parameters* - command line parameters you want to pass to your test executable. Note that for Test4Class parameters and host are configurable with help of template syntax (See Test DAL link above for more info), but for Test4Object they shall be specified explicitly.
- *exec_mode*, *init_timeout* and *init_depends_from* relationship are used to organize

synchronous and ordered sequences of tests per one object. See TestDAL link for more info.

**Run DVS**

As soon as tests binaries and Test Repository are OK, DVS can be used to test loaded configuration. DVS GUI can be started from the IGUI so use play_daq to start your partition:

- start the partition 'train_01' :

```
> play_daq train_01 no_obk
```

- When the IGUI appears, Boot the configuration and set it to the Running state.

Note that the partition must be *Booted* and *Running* in order for the tests to execute successfully.

- run DVS by clicking "Diagnostics" button on the bottom of the IGUI window.

You can also start DVS as a separate application from the command line using the dvs_gui utility. This utility takes the full path of the partition database file (or its relative path to where you are) after the "-d" command line flag as for the confdb_gui utility. The command line would look like:

```
> dvs_gui -d 'confdb_get_data_filename.sh'
```

**Load and Test Configuration**

When DVS window appears, it has already loaded your configuration. Select any component in the testable components tree at the left panel of the DVS GUI and push the '**test'** button to start testing and diagnostics inference for this component. To see the result and output of the test you have just implemented, select the component module 'Module_01' in the Hardware subtree.

Note that the test for the CPU Board can fail with the following error: "Computer CPU Board 'xxxxxx' is not running or has no remote shell (rsh) enabled". This is an error and indicates that rsh to your machine xxxxxx does not work. You should check this seperately and try again. If you are using ssh instead of rsh (by setting the **TDAQ_RSHELL_CMD** environment variable to ssh), note that this test still uses rsh exclusively, and will therefore likely fail.

# Chapter 5

# On-line Monitoring

This part of the exercise explains how to develop an event sampler example in C++ and a monitoring task example in Java, using the Online Monitoring system component of the ATLAS Online Software. This exercise is useful to anyone going either to implement an event sampler application that is responsible for supplying events to the event distribution sub-system or to develop a monitoring task that reads event from the event distribution.

## The On-line Monitoring system

The Online Monitoring system is responsible for the event transportation from event samplers providing event fragment sources up to the users' monitoring tasks. The system consists of the following sub-systems:

- Event Sampling, which is responsible for sampling event data flowing through the DAQ system and transportation of these events to the event distribution sub-system, each event sampler with responsibility for one crate of the DAQ system;
- User Monitoring task, which can request event fragments or full events with particular characteristics from the event distribution sub-system using it's public API;
- Event Distribution, which has a scalable architecture in order to be able to provide reasonable event transportation performance independently of the size of the DAQ system itself and a number of monitoring task working concurrently.

For more detailed information see User's Guide for Online Monitoring: http://atddoc.cern.ch/Atlas/Notes/157/mon-ug.html

## Event Sampler

The event sampler application is responsible for the communication with the data flow sub-system. It's implementation is specific for the different sub-detectors and DAQ crate types (e.g. ROD, ROC, SFC). The monitoring package provides only a skeleton class that defines an interface to the event distribution sub-system. A user wishing to carry out event sampling on his hardware must overload the methods in a "User" class which inherits from the Monitoring::EventSampler class. The user's

class that inherits from it is called MyEventSampler. Illustration 5.1 shows how to declare it.

```
class MyEventSampler: public Monitoring::EventSampler
{
public:
  MyEventSampler( const IPCPartition & p, const char *
detector_id, const char * crate_id );
  virtual Monitoring::Status startSampling (
    const Monitoring::SamplingAddress & ,
    const Monitoring::SelectionCriteria & ,
    const Monitoring::SampleAll & ,
    Monitoring::EventAccumulator * );
  virtual Monitoring::Status stopSampling (
    const Monitoring::SamplingAddress & ,
    const Monitoring::SelectionCriteria & );
  virtual void destroySampler ( );
};
```

Illustration 5.1 MyEventSampler class declaration (C++)

**Monitoring Task**

The monitoring task has to be define from which part of the DAQ system the events must be taken and identify certain event characteristics used to select events. In other words, using the monitoring system's terms and definitions, it is necessary to specify the events' Sampling Address and Selection Criteria. Illustration 5.2 shows how to pass the Sampling Address and Selection Criteria to the event distribution

and get back the Event Iterator's reference using the select method of the Monitoring class.

```
SamplingAddress sa = new SamplingAddress();
SelectionCriteria sc = new SelectionCriteria();
try {
  if ( args.length > 0 )
    sa.sa_detector = args[0];
  if ( args.length > 1 )
    sa.sa_crate = args[1];
  if ( args.length > 2 )
    sa.sa_module = args[2];
  if ( args.length > 3 )
    sc.sc_detector_type = Integer.parseInt( args[3] );
  if ( args.length > 4 )
    sc.sc_trigger_type = Integer.parseInt( args[4] );
  if ( args.length > 5 )
    sc.sc_trigger_state = Integer.parseInt( args[5] );
  if ( args.length > 6 )
    sc.sc_status_word = Integer.parseInt( args[6] );
}
catch( NumberFormatException e ){
  System.err.println( "ERROR:: Bad arguments " );
  return;
}
```

Illustration 5.2 Defining sampling address and selection criteria (Java)

Once the address and criteria have been defined it is necessary to create the MonitoringDistributor object and ask it to build the EventIterator for the these address and criteria. Illustration 5.3 shows how to do this.

```
Partition p;
if ( args.length == 8 )
  p = new Partition( args[7] );
Monitoring.Distributor ed = new Monitoring.Distributor(p);
Monitoring.Iterator ei;
try{
  ei = ed.select( sa, sc, false );
}
catch( Monitoring.BadAddress e ){
  System.err.println("ERROR:: Bad Address: detector " +
sa.sa_detector + ", crate " + sa.sa_crate + ", module " +
sa.sa_module );
  return;
}
catch ( Monitoring.BadCriteria e ){
  System.err.println( "ERROR:: Bad Criteria: detector_type
" + sc.sc_detector_type + ", trigger_type " +
sc.sc_trigger_type + ", trigger_state " +
sc.sc_trigger_state + ", status_word " + sc.sc_status_word
);
  return;
}
catch ( Monitoring.NoResources e ){
  System.err.println( "ERROR:: No resources" );
  return;
}
catch ( Monitoring.NoDistributor e ){
  System.err.println( "ERROR:: No distributor in partition
" + p.getName() );
  return;
}
```

Illustration 5.3 Creating event iterator (Java)

The instance of the EventIterator class that has been returned by the select method can be used to access events. Illustration 5.4 shows how to do this.

```
// Getting 100 events
int i = 0;
while( i < 100 ){
  int[] event;
  try{
    event = ei.try_next_event();
  }
  catch( Monitoring.NoMoreEvents e){
    continue;
  }
  i++;
  System.out.println( "Event " + i + " received. Size is "
+ event.length + "." );

  for ( int j = 0; j < event.length; j++ ){
    System.out.print( event[j] + " " );
  }
  System.out.println();
}
// If event iterator is not necessary anymore, destroy it.
ei.destroy();
```

Illustration 5.4 Getting events (Java)

**Monitoring exercise**

The monitoring test exercise will help you to develop an event sampler example in C++ which supplies events (from EventFragment.data data file, which you can find it in the training / monitoring / data subdirectory) to the event distribution sub-system and a monitoring task example in Java, that reads events from the event distribution. The exercise will use the same databases (which you can find in training / databases subdirectory) as all the other exercises do, and therefore the same partition train_01.

In the training / monitoring / cpp subdirectory you have all the C++ files you need to provide the executable file for the event sampler. In the training / monitoring / java you will find the java file for monitoring task. The solutions of the exercises are in the corresponding solution subdirectories.

You will build the executable for both event sampler and monitoring task . You will modify the databases in order to start the event sampler application by DSA supervisor (by means of play_daq script). The solutions for databases are in the training / monitoring / databases / solution subdirectory. You will check the functionality of the event sampler by using the Event Dump application. You will execute the monitoring task to get the events.

**Modifying the source files for event sampler**

All the needed files to provide the events sampler application you have it in the training / monitoring / cpp subdirectory: My_event_sampler_impl.h, My_event_sampler_main.cc and My_event_sampler_impl.cc. In the same subdirectory are all the needed makefiles.

- **Modify the source file My_event_sampler_main.cc to create the instance of the MyEventSampler class.**

**Modifying the source files for monitoring task**

The file needed to provide the monitoring task application, MonitoringTask.java, is in the training / monitoring / java subdirectory.

- **Modify the source file MonitoringTask.java to declare and initialise the sampling address and selection citeria variables**
- **Modify the source file MonitoringTask.java to call the select method of the MonitoringDistributor class**

**Building the event sampler**

You should have now all the source code necessary for building the event sampler example application. Change to training / monitoring / cpp subdirectory. If you have already installed the Online Software release and configured all the environment variables you need to build training code against the release on your platform (sourcing the training configuration script in the training directory), you can now build the event sampler using the makefile provided in the package:

```
> make     # compile and link the event sampler
```

As a result, you will have in the same directory the My_monitoring_sampler executable file.

**Building the monitoring task**

You should have now the source code necessary for building the monitoring task example application. Change to training / monitoring / java subdirectory. If you have already installed the Online Software release and configured all the environment variables you need to build training code against the release on your platform (sourcing the training configuration script in the training directory), you can now build the monitoring task. First you have to check that the environment variable CLASSPATH includes the path ${RELEASE_DIR} / share / lib / ipc.jar and ${RELEASE_DIR} / share / lib / Monitoring.jar, and add them if necessary.

```
> export
CLASSPATH=${RELEASE_DIR}/share/lib/ipc.jar:${RELEASE_DIR}/share/lib/Mon
itoring.jar:${CLASSPATH}
```

You can now build the executable file for monitoring task:

```
> javac  MonitoringTask.java
```

As a result, you will have in the same directory the MonitoringTask.class file.

**Modifying the databases**

The exercise uses the same databases (which you can find it in training / databases subdirectory) as all the other exercises do, and therefore the same partition train_01.

You have to modify the train_01.data.xml and train_01.sw.data.xml databases from training / databases subdirectory, in order to start the event sampler application by DSA Supervisor. If you have already installed the Online Software release and

configured all the environment variables you need, you have to copy first these two databases in a safe place, just in case. Perform the following steps:

- **Using the confdb_gui, as has been explained in other chapters of this document, define first in the train_01.sw.data.xml database your event sampler executable file as a new object of the class Program. Then create the corresponding object of the SoftwareObject class and create the relationship between them;**
- **Using the confdb_gui, define in the train_01.data.xml database a new object of the Application class calling it the event_sampler for example. It must has relationship with the respective instance of the SoftwareObject class;**
- **Define also the following attributes of the Application object:**
    1. **“Parameters” - the command line parameters for the event sampler: “-p train_01 -d Detector_01 -c ROCCrate01 -S 1024 -N 1 -D 100000 -F ${MY_PATH}/monitoring/data/EventFragment.data” (that means your application will simulate event sampling for the crate ROCCrate01 of the detector Detector_01),**
    2. **“InitTimeout” (which should be 0),**
    3. **“Creation Type” (which should be in our case Supervised),**
    4. **“Runs on” (your workstation),**
    5. **“Shutdown depends from” (the controller provided in the controller exercise) for this object.**
- **Select the Application class which you have just created for the Monitoring sampler with the right mouse button and select "Copy reference" on the menu which pops up. Right click on the partition and select "Link to ..." on the menu. Then select the option "Append to relationship ’Contains’". This will link the application to the partition and means that it will be run when the partition is started.**

### Testing the monitoring exercise

You can check now the event sampler and monitoring task are working.

Open a window and if you have already set up the Online Software release and configured all the environment variables you need, proceed the same as you did for testing the controller exercise: boot, load, configure and start the train_01 partition. If everything is OK you should be able to check the functionality of the controller as it is mentioned in the controller exercise. Besides, in the PMG panel of the IGUI, when you select the PMG agent you should see your event sampler in the panel as a running process.

When the partition is in the Running state, select the "Event Dump" button at the bottom of the IGUI window. In the Event Dupmp window select “Event Selection”. When another window appears select the partition, detector, crate and type ’Module_01’ in the module field. Then select the "Dump" button at the bottom of that window. You will see the event dump of one event in the window. On the left hand side the event is broken down into a tree structure of subdetectors, ROCs, ROBs, and RODs. On the right hand side you can see the raw data of the event. In

the Monitor panel of the IGUI you will see the statistics for the event sampler, which proves it has sampled one event and passed it on to the event dump.

Open another window, setup the Online Software release and configure all the environment variables you need using.training.sh script, check the environment variable CLASSPATH includes the path for the ${RELEASE_DIR}/share/lib/ipc.jar and for the ${RELEASE_DIR}/share/lib/Monitoring.jar, and add them if necessary

```
>export
CLASSPATH=${RELEASE_DIR}/share/lib/ipc.jar:${RELEASE_DIR}/share/lib/Mon
itoring.jar:${CLASSPATH}
```

Start your monitoring task application using the command:

```
>java -Dipc.ref.file=$IPC_REF_FILE MonitoringTask Detector_01
ROCCrate01 Module_01 1 0 0 0 train_01
```

You should see the 100 events printed out.

# Chapter 6

# Online Histogramming

This part of the exercise explains how to use the Online Histogramming subsystem. You will learn how to write a histogram provider and a histogram display using C++.

**The Online Histogramming subsystem**

The Online Histogramming (later refered to as OH) subsystem is one of the components of the ATLAS Online Software. The OH is a framework for histogram transportation in the distributed environment. It is responsible for the communication between two types of user applications: Histogram Providers (later refered to as HP) and User Histogram Task (later refered to as UHT).

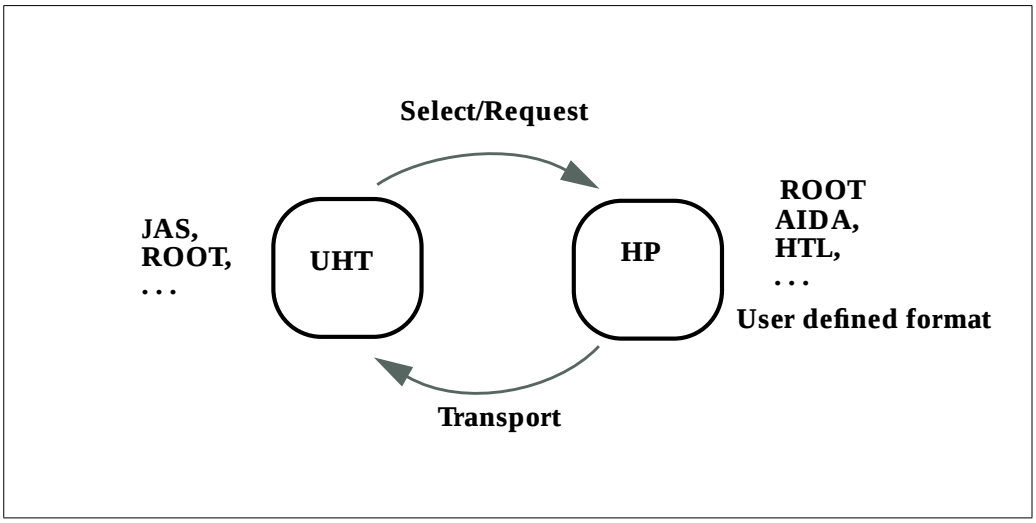



Illustration 6.1 Communication between Histogram Providers and User Histogram Task

**Histogram Provider**

A Histogram Provider is an application which may use one of the histogram filling frameworks like ROOT (An Object-Oriented Data Analysis Framework), AIDA (Abstract Interface for Data Analysis), HTL (histogramming Template Library), or any other means of building histograms. The HP Interface also supports export of histograms to the OH from a user defined format.

HP could be a user monitoring or analysis task, or a task providing histograms. When the histogram is ready the HP can make it publicly available by publishing it in the Online Histogramming system. The OH assigns a unique identifier to this histogram.

**User Histogramming Tasks**

A User Histogramming Task can access any histogram in the Online Histgramming system using an unique identifier. It is possible to enumerate all the histograms available in the OH system.

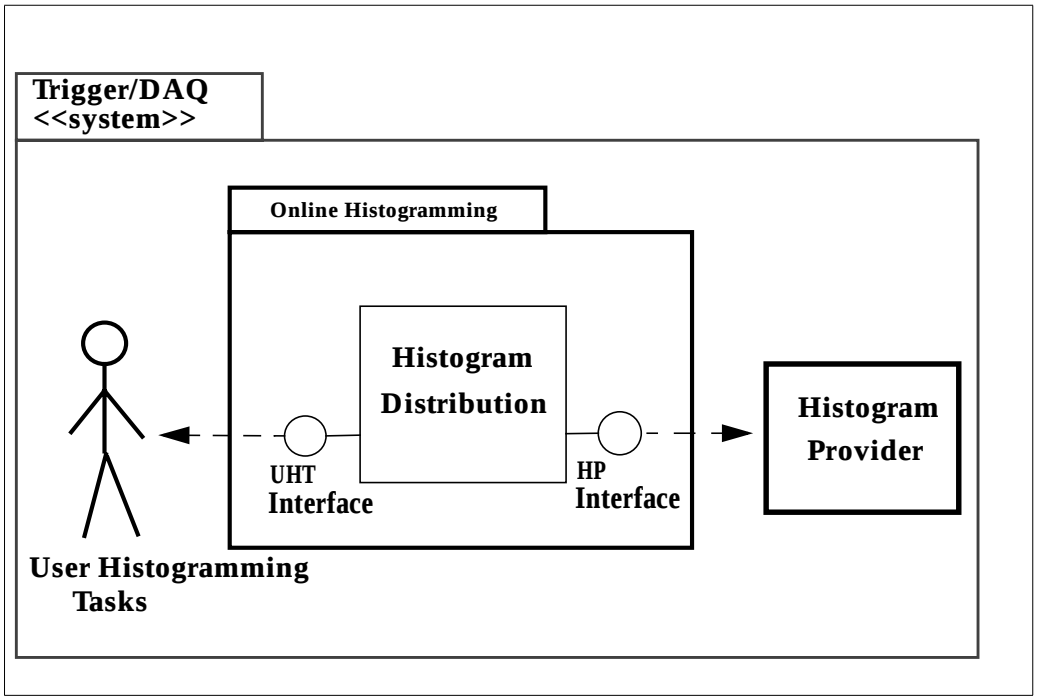



Illustration 6.2 Diagram of the Online Histogramming subsystem

**OH Interfaces**

**Histogram Provider** - sends histograms to the OH assigning a name to it;

**Histogram Receiver** - gets an histogram by name from the OH;

**Histogram Subscriber** - is notified when an histogram appears in the OH;

**Histogram Iterator** - enumerates all the histograms in the OH;

**Server Iterator** - enumerates all the existing OH servers;

**Provider Iterator** - enumerates all the active Histogram Providers.

**Online Histogramming Web Page**

The Online Histogramming Web Page contains references to the Requirements, User's and Developer's guide documents:

**http://atddoc.cern.ch/Atlas/DaqSoft/components/histogramming/Welcome.html**

**OH examples**

OH examples show how to use the OH. In the training we have two examples:

- **raw_provider** - shows how to write an histogram provider that exports histograms represented by arrays of some fundamental data type to the OH
- **root_display -** shows how to implement an application that imports histograms from the OH**.**

**Accessing the source code**

The source code of the OH examples is held in the histogramming directory. All necessary files are in two subdirectories:

```
raw_provider/
```

**raw_provider/**

**raw_provider.cxx -** a RAW provider example

**Makefile** - makefile to compile and link the example provider

**root_display/**

**root_display.cxx -** a simple utility which display histograms published in the OH using the ROOT framework

**Makefile** - makefile to compile and link the example display

**raw_provider** To view and modify the raw_provider's source code, open the raw_provider.cxx with your favourite text editor (e.g. nedit). This is shown on illustration 6.3.

We use a template class which provides the functionality to export histograms represented by arrays of some fundamental data type to the OH

```
template<class TContent, class TError, class TAxis, class TMap =
OHRawProviderDM> class OHRawProvider< TContent, TError, TAxis,
TMap >
```

The meaning of the template parameters are as follows:

- **TContent**: The data type used by user to represent bin contents (heights).
- **TError:** The data type used by user to represent bin errors.
- **TAxis:** The data type used by user to represent axis partitions.
- **TMap**: Governs how the user bins are accessed

```
//////////////////////////////////////////////////////////
//Create an OHRawProvider and publish some sample histograms //
//////////////////////////////////////////////////////////

IPCPartition p( partition_name );

//Here we choose the source format of the histogram data with
//the template arguments. Lets pretend we have 16-bit bin heights
// with 8-bit errors and floating point axes.

OHRawProvider<short,char,float> raw( p,(const char*)server_name,

                                    (const char*)provider_name);

if ( ! raw ) {

     cout << "The OH RAW provider was not successfully created,"
     << endl << "invalid arguments?" << endl;

     return 0;

}

// A sample annotation which stores the origin of the histograms

vector<string> labels, values;

labels.push_back( "Source" );

values.push_back( "OH RAW Provider example application" );

// Sample data, this should be retrieved from somewhere in a real
// app

short contents[11] = { 100,20,3,40,5,10,100,200,300,400,500};

char errors[11] = { 1,2,3,1,1,4,1,1,4,1,1};

float axis[11] = { 1,2,4,8,10,16,18,20,34,35,37};

// Publish a sample 1D histogram with variable width bins ,
//errors, underflow and overflow

   raw.publish( (const char*)histogram_name, "RAW Histogram 1D",
"X Axis", 7, axis, contents, errors, true, labels, values );

// Insert your code to publish a sample 2D histogram with
//variable width bins and errors here:

//...

}
```

Illustration 6.3 Create an OHRawProvider and publish histograms

To publish a sample 1D histogram we use the **publish** method ofOHRawProvider template class:

```
publish(const string & name,// Name describing the histogram

        const string & title,// The histogram title

        const string & label,// X Axis label

        long bincount,// The number of bins excluding underflow and
```

```
                    //overflow

            TAxis * axis,// A pointer to the axis partition (bincount + 1
                        //values)

            TContent * contents,// A pointer to the position where the bin
                              //contents (heights) are located

            TError * errors,// A pointer to the position where the bin
                           //errors are located or 0 if no errors

            bool outOfRangeBins,// Out of range bins or not

            const vector<string> & labels,// Labels for any annotations
                          //that should be attached to the histogram

            const vector<string> & values )// Values for any annotations
                           //that should be attached to the histogram
```

To publish a sample 2D histogram the **publish** method of OHRawProvider template class should be used:

```
publish(const string & name,// Name describing the histogram

        const string & title,// The histogram title

        const string & label,// X Axis label

        long xcount,// The number of bins along the x axis, excluding
                    //underflow and overflow

        TAxis * xaxis,// A pointer to the x axis partition (xcount + 1
                     //values)

        const string & ylabel,// Y Axis label

        long ycount,// The number of bins along the y axis, excluding
                    //underflow and overflow

        TAxis * yaxis,// A pointer to the y axis partition (ycount + 1
                     //values)

        TContent * contents,// A pointer to the position where the bin
                           //contents (heights)are located

        TError * errors,// A pointer to the position where the bin
                       //errors are located or 0 if no errors

        bool outOfRangeBins,// Out of range bins or not

        const vector<string> & labels,// Labels for any annotations
                           //that should be attached to the
                           histogram

        const vector<string> & values)// Values for any annotations
                         //that should be attached to the histogram
```

- **Modify the source code to publish 2D sample histogram.**

**How to build the raw_provider**

You should be in the raw_provider subdirectory. You can now build that provider using given Makefiles:

```
> make          # compile and link the raw_provider
```

**root_display**

To view and modify root_display's source code, open the root_display.cxx with your favourite text editor (e.g. nedit). It is shown on illustration 6.4.

- **Modify the source code to display histograms published in the OH.**

Fill the blank space in the program code in the following way:

- **The first thing that should be done, is to create a new OHHistogramIterator with name “it”. The constructor takes parameters as shown below :**

```
OHHistogramIterator::OHHistogramIterator ( IPCPartition & p,
                                           const string & server,
                                           const string & provider = ".*",
                                           const string & histoname = ".*",
                                           long year = ANY_YEAR,
                                           long month = ANY_MONTH,
                                           long day = ANY_DAY,
                                           long hour = ANY_HOUR,
                                           long minute = ANY_MINUTE,
                                           long second = ANY_SECOND
                                         )
```

***Parameters:***

***p*** - a valid IPCPartition

***server*** - name of a valid OH server

***provider*** - optional name of histogram provider, should only contain [a-z], [A-Z], [0-9], ’_’ and optional wildcars according to the egrep- style of regular expressions (see manual pages for egrep comand “ > man egrep”)

***histoname*** - optional name of histogram, should only contain [a-z], [A-Z], [0-9], ’_’ and optional wildcars according to the egrep- style of regular expressions (see manual pages for egrep comand “ > man egrep”)

***year*** - optional year when the histogram was published

***month*** - optional month when the histogram was published (JAN-DEC)

***day*** - optional day when the histogram was published (1-31)

***hour*** - optional hour when the histogram was published (0-23)

***minute*** - optional minute when the histogram was published (0-59)

***second*** - optional second when the histogram was published (0-59)

The iterator is a part of the mechanism that allows access to all histograms matching a given criteria (i. e. server name, provider name etc.)

The Iterator should be created with the following data: the partition, server name, provider name and histogram name.

- **As a next step check for the success of iterator creation (the boolean conversion is defined).**
- **Modify the source code by adding a loop over the histograms in the iterator. For each histogram display: name, provider, timeof creation and show histograms using retrieve() method :**

```
bool OHHistogramIterator::retrieve ( OHHistogramReceiver & receiver)
```

Retrieve current histogram.

***Parameters***:

***receiver*** - should be a user defined histogram receiver object derived from one of the OH receiver classes - **in our example we use MyReceiver class for that**: **class MyReceiver : public OHRootReceiver.** The constructor is **MyReceiver : (bool is_draw) : draw_( is_draw ) { ; }**

***Returns:*** true if the histogam was successfully received by the user, false if a communication error or other OH internal error occurs .

```
bool OHHistogramIterator::operator++ ( )
```

Advance the iterator one position.

***Returns***: true if new position is valid, otherwise false

Use the operator++ to get the next histograms.

If positioned at end false is returned by this operator.

```
string OHHistogramIterator::name ( ) const
```

Retrieve name of the current histogram.

***Returns***: the name of the histogram on the current position or "" if the current position is undefined.

```
string OHHistogramIterator::provider( ) const
```

Retrieve name of provider who published the current histogram.

***Returns***: the name of the provider who published the histogram on the current position or "" if the current position is undefined.

```
OWLTime OHHistogramIterator::time ( ) const
```

Retrieve time when the histogram was published.

***Returns***: the time when the histogram (set) on the current position was published or OWLTime( ) if the current position is undefined

- **Print the success/failure info of each display operation.**

The number of histograms in the iterator should be stored in the “count” variable. This variable is used later to recognise a situation when the iterator contains no histograms.

root_disply.cxx (fragments)

```
  ////////////////////////////////////////////////////////////////
  //Create an OHHistogramIterator object and retrieve all histogram
  //with the specified characteristics from the specified server
  ////////////////////////////////////////////////////////////////

      MyReceiver receiver( graphics );

      IPCPartition p( partition_name );

  //insert your code to create a OHHistogramIterator here:

  //...

  //check for the success of the OHHistogramIterator creation here:

  //...

      long count = 0;

  //insert your code thet displays basic information about histograms
  //in the OHHistogramIterator here:

  //...

      if ( count > 0 )

      {

      cout << "No more histograms available..." << endl;

      }

      else

      {

      cout << "No histograms found..." << endl;

      }

      if ( count > 0 && graphics )

      {

          cout << "Entering ROOT message loop..." << endl

               << "Click ’Quit ROOT’ on the file menu to quit"

               << endl;

          gSystem -> Run( );

      }

  }
```

Illustration 6.4 Create an OHHistogramIterator object and retrieve all histogram

**How to build the root_display**

Define ROOTSYS variable in your environment :

```
> export
ROOTSYS=/afs/cern.ch/sw/root/v3.03.09/rh72_gcc2953/root
```

You should be in root_display subdirectory. You can now build the root_display using the Makefile provided:

```
> make          # compile and link the root_display
```

**Testing the raw_provider and root_display**

When your raw_provider and root_display compile and link correctly you can publish histograms. The following steps must be done:

- **start anIPC partition with your partition name (e. g. mypartition):**

  ```
  > ipc_server -p partition_name &
  ```

- **start an IS (OH) server on your partition with your server_name (e. g. myserver):**

  ```
  > is_server -p partition_name -n server_name &
  ```

  Currently the OH server is equivalent to the IS server

- **Change directory to the raw_provider subdirectory. You can now publish your 1D and 2D histograms on your IS server using raw_provider:**

  ```
  > raw_provider -p partition_name -s server_name -n
    provider_name -h histogram_name
  ```

  (e. g. raw_provider -p myparition -s myserver -n myprovider -h myhisto)

- **Add $ROOTSYS/lib path to the LD_LIBRARY_PATH environment variable.**

- **Change directory to the root_display subdirectory. With root_display you can now display your histograms published in the OH (IS) server using the ROOT framework:**

  to display all histograms on a given server (e. g. myserver) in a given partition (e. g. myparition) use:

  ```
  > root_display -p partition_name -s server_name
  ```

  to display all histograms on a given server (e. g. myserver) in a given partition (e. g. myparition) which have been published by a specified provider named (e. g. myprovider) use:

  ```
  > root_display -p partition_name -s server_name -n
    provider_name
  ```

  to display all histograms on a given server (e. g. myserver) in a given partition (e. g. mypartition) which have been published by a specified provider named (e. g. myprovider) with histogram type name (e. g. myhisto) use:

  ```
  > root_display -p partition_name -s server_name -n
    provider_name -h histogram_name
  ```

  to display histograms in graphics mode you must add - g options e. g. :

```
> root_display -p partition_name -s server_name -n
  provider_name -h histogram_name -g
```

**The pictures below present the results of the commands presented above.**

```
bash: 0.0.16>root_display -p mypartition -s myserver -n myprovider
-h myhisto
Retreiving histogram myhisto created by myprovider at 5/3/02 10:22:53...
TH1.Print Name= Histogram1D_0, Entries= 0, Total sum= 378
fSumw[0]=100, x=-0.357143, error=1
fSumw[1]=20, x=1.5, error=2
fSumw[2]=3, x=3, error=3
fSumw[3]=40, x=6, error=1
fSumw[4]=5, x=9, error=1
fSumw[5]=10, x=13, error=4
fSumw[6]=100, x=17, error=1
fSumw[7]=200, x=19, error=1
fSumw[8]=300, x=21.3571, error=4
[OK]
Retreiving histogram myhisto created by myprovider at 5/3/02 10:22:53...
TH1.Print Name= Histogram2D_0, Entries= 0, Total sum= 778
fSumw[0][0]=0, x=-0.166667, y=-0.166667, error=0
fSumw[1][0]=0, x=1.5, y=-0.166667, error=0
fSumw[2][0]=0, x=3, y=-0.166667, error=0
fSumw[3][0]=0, x=6, y=-0.166667, error=0
fSumw[4][0]=0, x=9.16667, y=-0.166667, error=0
fSumw[0][1]=0, x=-0.166667, y=1.5, error=0
fSumw[1][1]=100, x=1.5, y=1.5, error=1
fSumw[2][1]=20, x=3, y=1.5, error=2
fSumw[3][1]=3, x=6, y=1.5, error=3
fSumw[4][1]=0, x=9.16667, y=1.5, error=0
fSumw[0][2]=0, x=-0.166667, y=3, error=0
fSumw[1][2]=40, x=1.5, y=3, error=1
fSumw[2][2]=5, x=3, y=3, error=1
fSumw[3][2]=10, x=6, y=3, error=4
fSumw[4][2]=0, x=9.16667, y=3, error=0
fSumw[0][3]=0, x=-0.166667, y=6, error=0
fSumw[1][3]=100, x=1.5, y=6, error=1
fSumw[2][3]=200, x=3, y=6, error=1
fSumw[3][3]=300, x=6, y=6, error=4
fSumw[4][3]=0, x=9.16667, y=6, error=0
fSumw[0][4]=0, x=-0.166667, y=9.16667, error=0
fSumw[1][4]=0, x=1.5, y=9.16667, error=0
fSumw[2][4]=0, x=3, y=9.16667, error=0
fSumw[3][4]=0, x=6, y=9.16667, error=0
fSumw[4][4]=0, x=9.16667, y=9.16667,
error=0
[OK]
No more histograms available...
```

Illustration 6.5 results of root_display usage

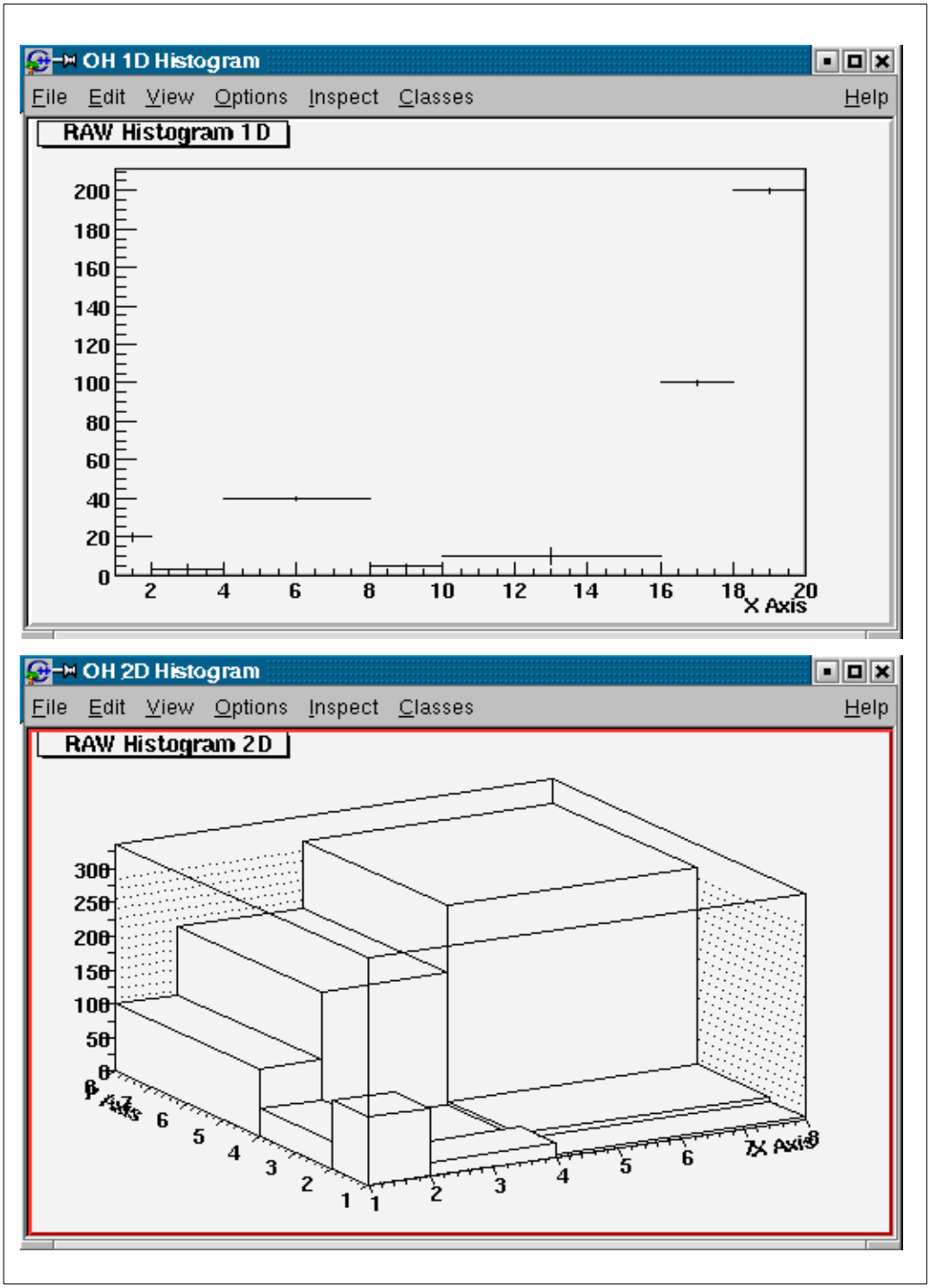



Illustration 6.6 results of graphics mode root_disply usage

# Chapter 7

# Resource Manager

This chapter of the exercises is dedicated to the usage of the Resource Manager. You will learn how to ask for resources, use resources and free them again using the Resource Manager Library in C++.

**The Resource Manager**

On large machines such as for example the ATLAS detector there are a lot of resources such as controllers, data-taking machines, graphical user interfaces and so forth which are useful for many purposes. It might happen quite often that more people or applications want to use a special device than this device can handle. (For example the controlling application of a specific part of a detector. Two people trying that at the same time might cause problems.) So the available resources have to be organized in such a way that the systems can work without any problems.

The Resource Manager is created for this task and allows an easy handling of various resources.

**How does the Resource Manager work?**

The Resource Manager (see Illustration 7.1) is divided into a client and a server part. The server covers all the necessary classes concerning the resource management. The dynamic database which handles resources and their various states is part of it and hidden in an internal class used by the Resource Manager Server.

The client class allows applications to ask for resources, to handle them and to free them again.

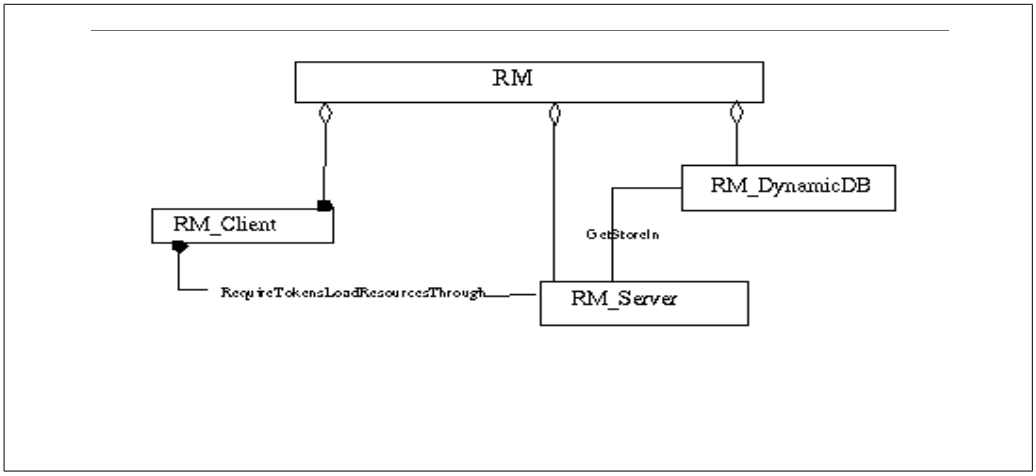



Illustration 7.1 The dialog for building a Software Resource

Shared and exclusive resources which have to be set up in the configuration databases are loaded into the dynamic database. Once this is done applications can use the Resource Manager Library to ask for resources. If they are granted the application gets back a so called token which is connected to the given resources and the application can use these tokens to communicate with and control the allocated resources. When the resource is not needed anymore the application can (and should of course) free the resource for the use of others again.

How many tokens can be used at once for a specific resource depends on the resource itself.

Applications may also load and unload partition resources into or from the dynamic database and ask for information about different tokens and resource states.

### Usage of the Resource Manager

As already mentioned the Resource Manager consists of a C++ library. It is part of the Online Software and its header file can be found in the Online software package under

**$TDAQ_INST_PATH/include/rm/RM_Client.h**

New classes are introduced of which some will be used during this chapter:

| Class | Functionality |
|---|---|
| TokenID | Contains the token number for various allocated resources. |
| AVAILABILITY | An enum object allowing the use of the states AVAILABLE, LOCKED, NOT-FOUND. |
| strTokenstatus | Contains the information if the operation was successful or not and if not what may be the reason. It returns one of the following strings: SUCCESS, UNSUC-CESS, FREE, ALLOCATED or INUSE. |
| RMERROR | Used by some methods to give more detailed information about what hap-pened during an operation. |
| RMInfo | Offers the possibility to interpret the information of various other methods in a convenient way. |
| RM_Client | Allows the handling of resources and tokens. |

Table 7.1 Classes in the RM_Client library

These classes and their methods can be used to handle resources which will be explained in more detail later. For information beyond this training one can consult the documentation of the resource manager available via

**http://atddoc.cern.ch/Atlas/DaqSoft/components/resmgr/Welcome.html**

**Adding Resources to the database**

As mentioned before resources have to be added to the databases to use them. Additionally one needs to create a Software Object that owns those resources as well as an application to which this object is linked and a program that is the implementation of the application. Finally the application must be linked to the partition so that the resources become part of the partition. To do all this the following steps are necessary.

One should modify the database for the partition **train_01** via the command

**> confdb_edit_data.sh -d train_01.data.xml**

when being in the directory **databases**. A window showing the schema files, the database files and the list of possible objects is opened. In this window one should set the **train_01.sw.data.xml** file to the active state. This makes sure that all changes are put into that file and not others.

The next step is to introduce resources in the database. This can be done by scrolling down the lowest part of the window until one can see the **SW_Resource** row. Activate it via a leftclick and build a new object via the menu that appears after a rightclick. Use the attributes given in illustration 7.1. Then build a second **SW_Resource** where every 1 is exchanged by a 2. (This includes the name and the ID as well as the number of available copies.) In the following the **ID** must always be the same as the **Name** is. So when in this subsection names for objects are given one should also use them for the **ID**'s.

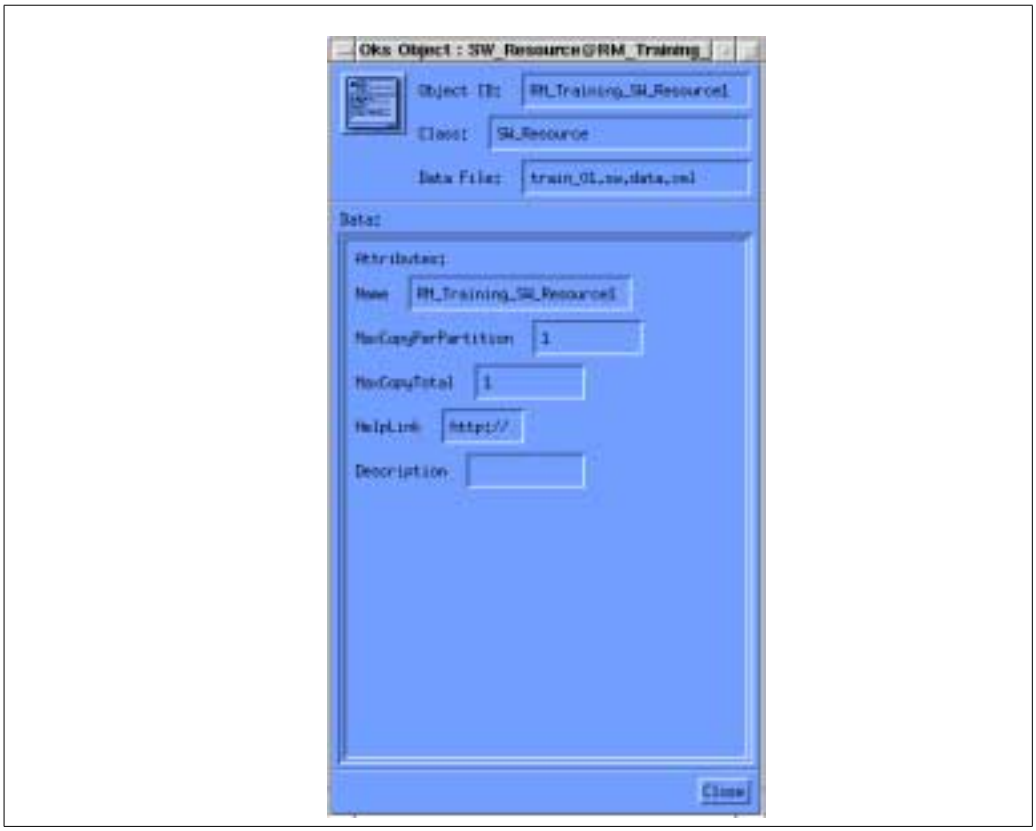



Illustration 7.2 The dialog for building a Software Resource

Now one must make a new **SW_Object** called **RM_Training_SW_Object** and link to the **NeedsShared** list of this **SW_Object** the two resources one just created. To put the two resources just created in the **NeedsShared** list keep this window open. Go back to the main window. Double click on the **SW_Resource** category, and a

window with both resources will appear. Right click and choose select on the menu. Go back to the **SW_Object** window and right click in the field **NeedsShared** and select the share item on the menu. Repeat for the other **SW_Object**.

One also needs an object of the **Program** class. It ought to be called **RM_Training_Program**, enter in the dialog for the executable file the line **${MY_PATH}/resources/RM** (this is the application that has to be written during this chapter) and establish a link from the **SW_Object** one just built to this **Program**. When this is done one additionally needs to make a link of the **Program** to the **ImplementedBy** list of the **SW_Object**.

The next step is to create the **Application** object called **RM_Training_Application**. The **CreationType** should be set to **Default** and two links must be set. One links the **RM_Training_SW_Object** to the **SW_Object** dialog and the other one is a link to **MyWorkstation** for the **RunsOn** list of the **Application**. The **InitTimeout** variable has to be set to **0**.

Finally the **Application** must be linked to the partition **train_01** in the **Contains** list.

This completes the necessary steps to introduce resources to a partition.

### Usage of the C++ library

During this chapter several methods to allocate resources and to get information are used. These are

RMInfo* **RM_Client::GetPartitionAllResInfo**(const char* **partition**, RMERROR ***errpub**)

Gets information about all resources of the Partition **partition** stored in the dynamic database.

AVAILIABILITY **RM_Client::GetPartitionStatus**(const char* **partition**, RMERROR ***errpub**)

Returns the information if **partition** is either AVAILABLE, LOCKED or NOTFOUND.

Tokenstatus **RM_Client::LockPartition**(const char* **partition**, RMERROR ***errpub**)

Locks **partition** so that all resources of that Partition become unavailable.

Tokenstatus **RM_Client::UnLockPartition**(const char* **partition**, RMERROR ***errpub**)

Unocks **partition** so that all resources of that Partition can be used when allocated.

TokenID **RM_Client::AskApplicationResources**(const char* **partition**,const char* **application**, RMERROR ***errpub**, const char* **clientname**)

Asks for all the resources linked with **application** in partition **partition**. **clientname** is optional and can be used for further identification. (e.g. one can free all resources of a client at once.) Returns token number if successful.

Tokenstatus **RM_Client::CheckAllocated**(const TokenID **tid**)

Checks if token with token number **tid** is allocated. Returns SUCCESS if it is allocated and UNSUCCESS if otherwise.

Tokenstatus **RM_Client::CheckCompleteValidity**(const TokenID **tid,**const char* **application**)

Checks if token with token number **tid** is allocated for the application **application**. Returns SUCCESS if it is allocated and UNSUCCESS if otherwise.

TokenID **RM_Client::AskResourcesDirectly**(const char* **partition**,const char* **application**,const char* **rnlist**, RMERROR ***errpub**, const char* **clientname**)

Asks for resources given in **rnlist** linked with **application** in partition **partition**. It returns token number if successful.

Tokenstatus **RM_Client::FreeAllPartitionResource**(const char* **partition**, RMERROR ***errpub**)

Frees all resources in partiton **partition**. Returns SUCCESS or UNSUCCESS depending on the result of the freeing.

RMInfo* **RM_Client::GetOneTokenInfo**(TokenID **tid**, RMERROR ***errpub**)

Gets information about all resources allocated with **tid**.

Tokenstatus **RM_Client::FreeResourcesByToken**(TokenID **tid**, RMERROR ***errpub**)

Frees all resources allocated with **tid**.

**What does the application do?**

The application one should implement does the following:

First the partition train_01 is locked, checked, unlocked and checked again. Then all the resources in RM_Training_Application are allocated and two checks are run. One simply checks, if the token with number tid is allocated and the second check additionally check if this also belongs to the application RM_Training_Application. Next information of just one resource is presented and once again the program tries to allocate all the resources of RM_Training_Application. This should fail since

RM_Training_SW_Resource1 may only be allocated once. The trial to allocate only the RM_Training_SW_Resource2 a second time succeeds whereas the third trial fails again. Then all resources of the partition train_01 are freed again. The second part of the application allocates just one resource and information about the token

under which it was allocated is presented and every resource being part of this token is freed again.

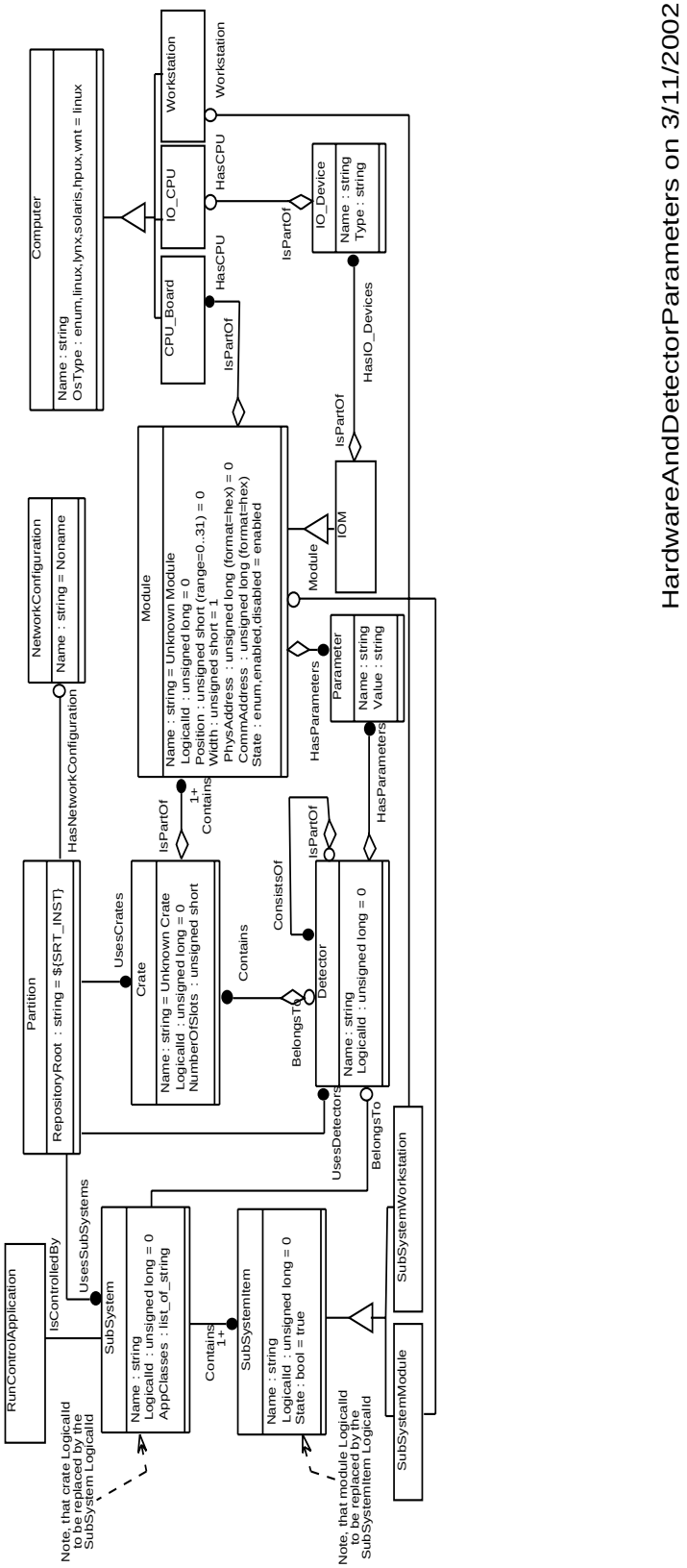
```
// Defining names for the application, resources and the
partition
  application=(char *) "RM_Training_Application";
  rnlist=(char *) "RM_Training_SW_Resource1";
  rnlist2=(char *) "RM_Training_SW_Resource2";
  dummy=(char *) "";
  const char* p="train_01";


cout << "Getting information from train_01" << endl;
  errpub = new RMERROR;
  tmpinfo = RMC.GetPartitionAllResInfo(p,errpub);
  tmpinfo->Info_res_table();
  delete errpub;


  cout << "Checking availibility of train_01" << endl;
  errpub = new RMERROR;
  avail=RMC.GetPartitionStatus(p,errpub);
  if (avail==AVAILABLE)
    tmpString="AVAILABLE";
  else if (avail==LOCKED)
    tmpString="LOCKED";
  else
    tmpString="NOTFOUND";
  cout << "Partiton status is "<< tmpString << endl;
  delete errpub;


  cout << "Trying to lock partition train_01" << endl;
// Enter your code for locking partition train_01 here
//........//
tid = RMC.AskApplicationResources(p,application,errpub);
  cout << "TokenID of AskApplicationResources = "<<  tid  <<
endl;
```

RM.cxx (fragments)

Illustration 7.3 Create an OHRawProvider and publish histograms

**Accessing and modifying the source code**

The C++ source code for this chapter can be found in the **resource** subdirectory and is named **RM.cxx**. One can use his or her favourite editor to modify the source code



in order to succeed in the following tasks.

- **Lock partition train_01 in the given section of the source code**
- **Ask for all resources that are linked to the application** RM_Training_Application **(one should use** tid **for the TokenID in order to keep the rest of program functional)**
- **Check if the token with TokenID** tid **is allocated and belongs to the Application** RM_Training_Application**.**
- **Free all resource that belong to the partition** train_01**.**
- **Get information about the resources that are allocated with the TokenID** tid**.**
- **Free resources that are allocated with the TokenID** tid**.**

**Compiling and testing of the application**

When the modification are done one can compile the source when being in the **resource** directory code with the command

**> make**

In order to test RM file one needs to run

**> play_daq train_01 no_obk &**

When the **IGUI** is in the running state one can run the test application via

**> RM**

when in directory **resources**. It should print out information to the console.

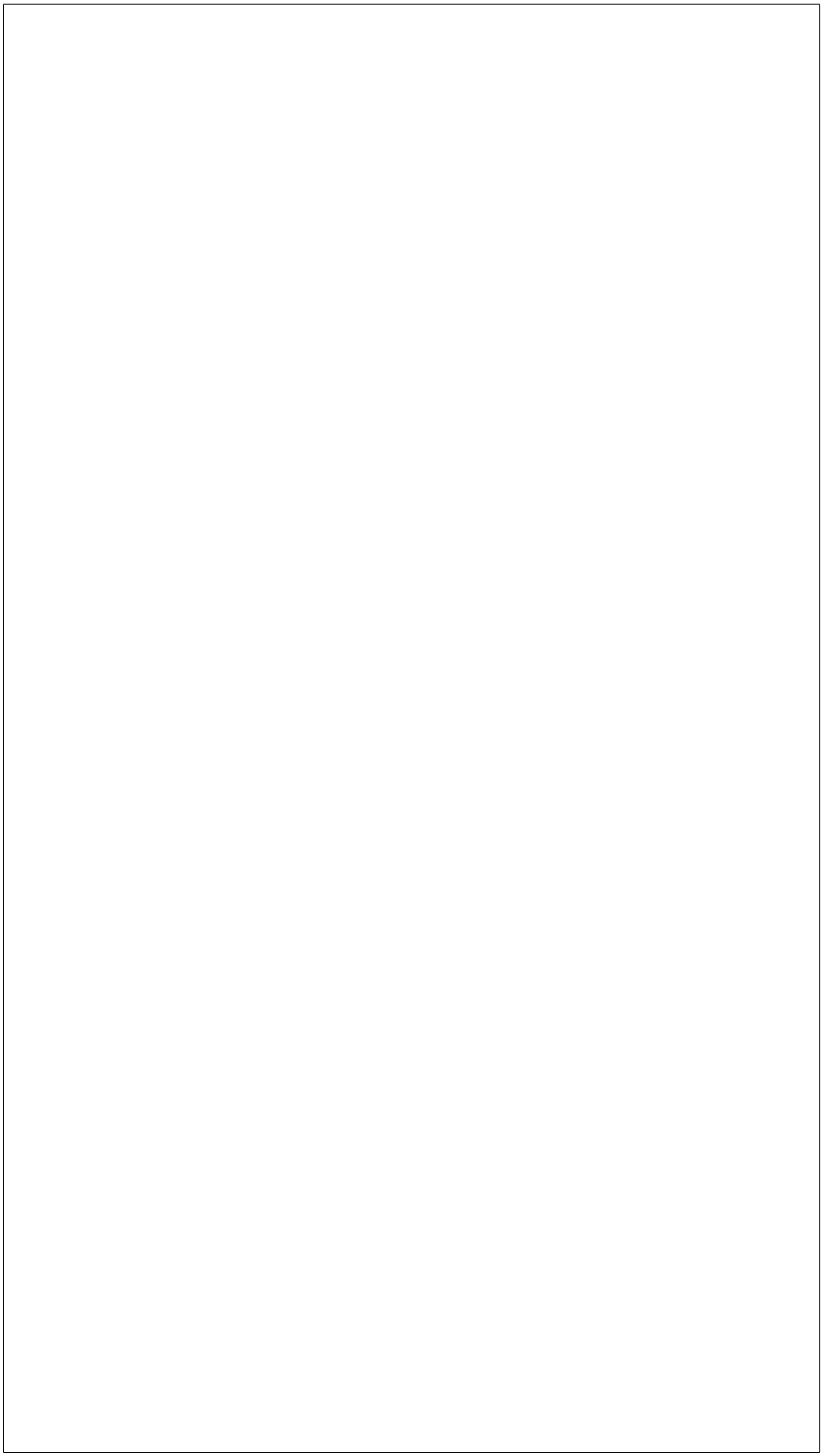


Illustration 7.4 Setup for the Hardware Database

**Hardware Resources**

Apart from Software Resources (all the resources up to now were Software Resources) there is also the possibility to use so called Hardware Resources. Software Resources are abstract entities. An application using a Software Resource can run anywhere and is not necessarily aware of which machine it runs on. The Software Resource only defines how often it can run in one partition or in general but not on which machines that are part of the partition.

Sometimes it is necessary to really know which device is used. One might want to us a specific crate or a specific detector that is part of the partition which is described in the databases. Due to the fact that one wants to describe a specific device a more precise configuration is necessary. One must describe the exact relationships starting from the Computer (Illustration 7.4) to the specific device that is meant to be a Hardware Resource in the database.

**Introducing Hardware Resources in the Database**

- **Introduce a Hardware Resource related to the Read-out Create of the partition** train_01 **in the database.**

This means that on all the machines that are part of the Read-out Crate the application may only run as often as allowed in the Hardware Resource specification.

To introduce the Hardware Resource one needs to use the **confdb_edit_data.sh** program to modify the database. The fist step is to change the **RunsOn** Parameter of the Application **RM_Training_Application** to the **VirtualCPU** which is set up in the database. This virtual CPU is the same machine as **MyWorkstation** but the virtual CPU is also part of the partition (which is necessary for handling the exact relationships). A Hardware Resource object named **RM_Training_HW_Resource_Crate** must be created. The Hardware Class is **ROC** for a **Read-out crate**. The **DB_Path** describes the relationship of the Hardware Resource to the **Computer** on which the application is running. Looking at Illustration 7.4 one sees that the relationship starting from the computer to the crate is **IsPartOf.IsPartOf**, since the **CPU_Board** (which is a **Computer**) is part of the **Module** and the **Module** is part of the **Crate**.

**Usage of Hardware Resources**

The objects and methods i.e. the whole usage of allocating, using and freeing Hardware Resources are the same as for Software Resources. The RM application that has been made is still usable and allocates the Hardware Resource as well. By running it you will see that there is only one slight change. The name of the Hardware Resource is not the name itself but the name plus the sign “@” followed by the name of the crate. In this case it means that the name of the Hardware Resource is **RM_Training_HW_Resource_Crate@ROCCrate01**.

**The rm_gui**

Apart from having to write a test application of one’s own there is another option to get to know the behaviour of resources, tokens, applications and their relationships. After having started the partition and the resource manager itself by hand or via the **play_daq** command simply use

**> rm_gui**

to start the resource manager gui. It is a front-end for some of the possible commands concerning resources. It is easy to use and allows for experimenting with tokens, resources and available information. It is strongly advised to use it until one feels comfortable with the concepts used by the resource manager.